UFR SCIENCES DES ORGANISATIONS

Centre de Recherche en Informatique Appliquée

THÉSE

Contribution à la Conception et l'Implantation de la Structure de Données Distribuée & Scalable à Haute Disponibilité LH*$_{RS}$

Pour obtenir le titre de

Docteur en Informatique

(Arrêté du 25 avril 2002)

Présentée et soutenue par

Rim MOUSSA

le 4 octobre 2004, devant le jury composé de :

Directeur de Thèse:	**Pr. Witold LITWIN**
	Professeur à l'Université Paris Dauphine.
Rapporteurs:	**Pr. Thomas J. E. SCHWARZ, S.J.**
	Professeur Associé à l'Université Santa Clara, USA..
	Pr. Toré RICH
	Professeur à l'Université d'Uppsala, SUEDE.
Suffrageant:	**Pr. Gérard LÉVY**
	Professeur émérite à l'Université Paris Dauphine.

L'université n'entend donner aucune approbation ni improbation aux opinions émises dans cette thèse : ces opinions doivent être considérées comme propres à leur auteur.

Dédicaces

A mes chers parents Hasna & Ali

Remerciements

Je remercie tout particulièrement :

Pr. Witold Litwin, professeur à l'Université Paris Dauphine et directeur du CERIA, de m'avoir accueillie au sein de l'équipe CERIA et de m'avoir encadrée en thèse. Je lui adresse mes remerciements pour ses encouragements incessants, sa disponibilité constante, la pertinence de ses jugements, et la sollicitude dont il m'a entourée. Merci pour les conseils, qui ont permis d'améliorer ce manuscrit.

Pr. Gérard Lévy, professeur à l'Université Paris Dauphine, n'oubliant certainement pas le soutien moral, les encouragements, que vous m'avez donnée, et qui m'ont motivé dans mon travail au CERIA. Merci pour les conseils, qui ont permis d'améliorer ce manuscrit.

Pr. Thomas Schwarz, professeur à l'Université Santa Clara aux Etats Unis d'Amérique. Merci pour les conseils éclairés. Je n'oublierai certainement pas les discussions fructueuses qui ont rendu possible la soutenance de cette thèse. Merci pour les conseils, qui ont permis d'améliorer ce manuscrit.

Pr. Toré Risch, professeur à l'Université d'Uppsala en Suède et directeur du Lab. UDBL, pour le vif intérêt qu'il manifeste à l'égard de mes travaux de recherche, et ses remarques constructives et encourageantes.

Les coordinateurs de Microsoft Corp., particulièrement, Dr. Jim Gray, pour l'intérêt qu'ils portent à mon travail.

M. Samba Ndiaye maître assistant à l'université Cheikh Anta Diop de Dakar, M. Djamel Eddine Zegour professeur au CNI d'Alger, M. Mohamed Tidiane Seck maître assistant à l'Ecole Polytechnique de Dakar, M. Enrico Nardelli professeur à l'Université de l'Aquila en Italie, à qui j'ai exposé mes travaux de recherche, et m'ont enrichie par leurs critiques constructives.

Le ministère de l'enseignement supérieur tunisien de m'avoir accordée une bourse de mérite pour poursuivre des études doctorales en France.

Tous les enseignants qui ont participé à ma formation, de la maternelle à l'université, et ont fait de moi ce que je suis aujourd'hui.

La commission de spécialistes de l'Université Paris Dauphine de m'avoir accordée un poste ATER, m'ayant permis de renforcer ma vocation pour l'enseignement et de m'assurer un financement digne de thèse.

Tous les professeurs et responsables d'UFRs, qui m'ont fait découvrir l'enseignement et permis d'enseigner, notamment, Mme F. Semmak, M. F. Boufares. Mme J. de la Bruslerie, M. B. Goldfarb et M. J. M. Janod.

Cette thèse n'aurait pas pu être menée à bien sans les membres de l'équipe CERIA. J'adresse une pensée particulièrement amicale à Fethi, Yakham, Aly, Soror, Fatma et Riad, pour leurs encouragements, bienveillance, leur bonne humeur, qui m'ont aidé à mieux travailler.

Les assistantes administratives du CERIA, Mme Annie Crimmersmois et Mme Janine Verepla, pour leur disponibilité, écoute et coordination avec les services administratifs de Dauphine.

Mes parents de m'avoir toujours soutenue dans mes choix d'études et professionnels. Je remercie mon frère Samy et ma petite sœur Naouel pour leur soutien et la patience dont ils ont fait preuve à mon égard, tout au long de l'élaboration de ce mémoire, et l'effort pour que je jouisse de chaque moment des vacances qu'on passe ensemble.

Enfin, à tous mes amis qui m'ont assistée et encouragée, avec une pensée à Sondes et Manel.

Résumé

Le thème de recherche de cette thèse concerne la conception et la mise au point d'une structure de données distribuée et scalable (SDDS) à haute disponibilité : LH^*_{RS}. Cette structure permet de mémoriser un très grand fichier de données. Les données du fichier LH^*_{RS} sont réparties par le hachage linéaire distribué (LH*) sur les nœuds de stockage connectés par un réseau. Le nombre des nœuds augmente dynamiquement avec la taille du fichier. La distribution est transparente pour l'application. LH^*_{RS} tolère, en plus de LH*, l'indisponibilité de $k \geq 1$ nœuds de stockage de données. La valeur de k peut être paramétrée à la création du fichier et peut augmenter automatiquement avec les insertions (autrement la fiabilité du fichier aurait nécessairement décru). La tolérance résulte de l'ajout de données de parité, encodées selon un nouveau code que cette Thèse contribue à proposer, de type Reed Solomon (RS). Cette classe de codes est particulièrement intéressante dans notre cas par son caractère systématique et MDS (ang. *Maximal Distance Separable*). La propriété caractéristique de notre code est que la matrice de parité contient une ligne et une colonne de '1's. Il en suit l'utilisation la plus extensive à l'heure actuelle et à notre connaissance dans cette classe de codes, de XOR (l'addition dans les corps de Galois). Pour le cas le plus fréquent de l'indisponibilité d'un seul nœud, notre code est dès lors aussi rapide que les schémas populaires de type RAID, limités toutefois à $k = 1$. A ceci s'ajoute l'utilisation de la multiplication par une table de logarithmes dans le corps de Galois usité qui est $CG(2^{16})$. Ce qui a conduit l'encodage et le décodage par une matrice de parité logarithmique, dont les coefficients sont les logarithmes de ceux de la matrice de parité précitée. L'ensemble des optimisations conduit à l'encodage et à la récupération de données particulièrement efficaces, comme la Thèse le démontre.

Nos résultats contribuent à la conception du code proposé à travers l'étude théorique et expérimentale de diverses techniques de codage/ décodage candidates pour LH^*_{RS}, depuis la proposition de cette SDDS en 2000. L'étude expérimentale concernait tout particulièrement la conception et l'implémentation d'un gestionnaire prototype pour cette SDDS.

Nous avons montré l'efficacité de notre gestionnaire au niveau du temps des opérations de recherche et de mise à jour de données. Nous avons montré aussi

l'efficacité de la récupération de données mémorisées sur un ou plusieurs nœuds de stockage devenus indisponibles, jusqu'à tout *k* utile en pratique. La création de notre prototype a été notamment la seule voie pour estimer la vitesse réelle de ces opérations. La complexité du système ne permettait pas une étude exclusivement théorique.

Nous avons basé notre gestionnaire sur la version 2000 de LH^*_{RS} [L00, M00]. Tout au long de la thèse, en partant de la version 2000, nous avons réalisé ensuite un cycle itératif de conception, implémentation et l'évaluation expérimentale de performances. Notre but constant était la minimisation du temps d'exécution de manipulations impliquant le calcul de parité. Les améliorations successives touchaient d'une part le code usité ainsi que le corps de Galois employé, pour minimiser le temps CPU. Elles touchaient également la composante réseau, pour minimiser le temps de communication.

Ainsi, nous avons d'abord accéléré et rendu plus fiable la propagation de mises à jour aux serveurs de parité. Notamment, en changeant le protocole de communication d'UDP à TCP/IP. Puis, nous avons introduit une matrice de parité ayant la 1ère colonne de 1's, pour accélérer le codage et le décodage. Ensuite, nous avons accéléré la vitesse d'opération d'éclatement et de récupération par une nouvelle architecture de communication avec les connexions TCP/IP ouvertes en mode passif. Puis, nous sommes passés à notre matrice de parité finale, ayant aussi la 1ère ligne de 1's. Ensuite, nous sommes passé du corps de Galois $CG(2^8)$ utilisé initialement à $CG(2^{16})$ et nous avons prouvé que ce changement était également utile. Puis, nous avons ajouté le contrôle de flux, après que les expériences aient montré la possibilité de perte d'enregistrement, dans le cas d'insertion en mode groupé, (ang. *bulk*). Enfin, pour le prototype final, nous avons amélioré le dynamisme de la procédure d'ajout d'un serveur libre, (ang. *spare*), pour la reprise ou l'extension du fichier durant un éclatement.

Au final, les résultas d'accès et de récupération, obtenus sont efficaces. Nos expérimentations avec un fichier de 125K d'enregistrements et un groupe de quatre serveurs de données, montrent que notre prototype récupère un serveur de données de 3,125 Mo en moins d'une demi-seconde, et trois serveurs (9,375 Mo) en 1,2 sec. Les vitesses de récupération obtenues sont de 5.66 MB/s, 7,14 MB/s, 7,89 MB/s respectivement pour la récupération de 1, 2, 3 serveurs de données. Le temps d'exécution

d'une requête de recherche, d'insertion ou de mise à jour est au maximum égal à 0,5msec dans un fichier 3-disponible. Les performances sont également dues au stockage des données en RAM.

Nos travaux ont donné lieu à trois publications [ML02, LMS04, MS04, LMS04-*b*]. Le prototype a notamment été montré à VLDB 2004 [LMS04].

Nos résultats prouvent la faisabilité d'une SDDS à haute disponibilité. Ils ouvrent aussi diverses perspectives de recherche. Il s'agit d'abord de la recherche d'autres codes et des optimisations de codage et décodage. Puis, des améliorations d'ordre architectural, telle que l'exécution d'appels de communication non bloquants. Enfin, on devrait étudier des applications de nos résultats, notamment aux SGBDs à haute disponibilité, les systèmes P2P et aux grilles de données.

Extended Abstract

Contribution to the Design and Implementation of the Highly Available Scalable Distributed Data Structure LH*$_{RS}$

Thesis Goal

This thesis studied the design and implementation of the scalable and distributed data structure (SDDS) termed LH*$_{RS}$. LH*$_{RS}$ distributes scalable data on storage nodes using the distributed linear hashing schema LH*. In addition it tolerates the unavailability of any $k \geq 1$ unavailable nodes. The value of k can be large enough for any practical need. It can also scale transparently with the file, to prevent the reliability decrease, otherwise necessary. The scheme uses the parity calculus, to minimize the storage overhead. The calculus has no impact on the search time. We seek in this context to minimize the time of the update and recovery operations.

Thesis Contribution

Our theoretical results contribute to the definition of a particularly efficient parity calculus for our scheme. It uses a novel parity data calculus by a specific Reed Solomon Code (RS codes). The choice of RS codes results from the fact that are MDS and systematic codes. To tune the access performance in this context, we designed and implemented the prototype LH*$_{RS}$ Manager. Our system allows for the efficient record search and insert operations as well as for the recovery of unavailable data for any practical value of k.

We validated our choices by extensive experimental performance analysis. This was in particular the only way to estimate the actual performance of our scheme, as well as the contribution of various improvements, introduced over the time of our study. The complexity of the underlying implementation prohibited in practice any purely theoretical conclusions.

To obtain our results, we carried out throughout the thesis, cycles of design, implementation and experimental validation, as we detail in what follows. Our constant purpose was best access and recovery performance. The results contributed, on the one hand, to the evolution of the parity calculus for the scheme. The final parity matrix proposed for LH*$_{RS}$ has 1st column and 1st row of '1's. This leads to more frequent use of XORing than in the initial matrix of LH*$_{RS}$, [LS00]. In particular, for $k = 1$, our schema performs XOR encoding/decoding, and behaves as the most widely used RAID systems. We have further shown the utility of using the logarithmic matrix derived from the above one (for both encoding and decoding). Finally, while the initial scheme was foreseen for the Galois Field GF(2^8), we have shown that GF(2^{16}) should be used instead, as more efficient in our case.

We have further contributed progressively to the optimized design of main architectural components of our prototype. In particular, we progressively improved our network component. We provided it with a more efficient TCP/IP connections handler, where TCP/IP connections are passive OPEN [ISI81, MB00]. We have also designed the flow control and acknowledgements management strategy based on the principle of message conservation until delivery [JK88, GRS97, D01]. Finally, we added a dynamic addressing structure, updated through multicast probe for new data/parity servers.

As we already mentioned, we used the experimental performance analysis to prove the validity of our choices. We measured every improvement to find out its relative and absolute incidence on the response time. The final results show attractive access and recovery performance. Our testbed files with 125K records, recovered in almost half a second from a single unavailability and in about 1.2 seconds from a triple one. Our prototype recovers a data bucket group of size $m = 4$ from 1-unavailability at the rate (speed) of 5.66 MB/s of data. Next, we recover 2 data buckets of the group at the rate of 7.14 MB/s. Finally, we recover the group from 3-unavailability at the rate of 7.89 MBs. Individual search, insert and update times were at most 0.5 msec for a 3-available file. The performance is also due to the data storage in the distributed RAM. These results prove the efficiency of our scheme.

Future work should focus on the investigation of efficient erasure-resilient codes. It should also be possible to improve our server speed by using communication APIs calls in the background (by using I/O completion ports). We also foresee the study of applications of our scheme, to a high-availability DBMS design especially.

Publications

We have published four papers about our work [ML02, MS04, LMS04, LMS04*b*]. The latter publication, VLDB-04, corresponds also to the demo of our system.

In [ML02], we report the design and performance results of our first LH*$_{RS}$ prototype. The prototype implementation extends the one of [L00, M00], but improves the performances, essentially by replacing UDP by TCP/IP for parity update propagation during server split, parity bucket creation and bucket recovery. In this paper the servers' architecture is inherited from the work of [B00, D01], and basic encoding/decoding routines using Reed Solomon codes without any optimisations.

In [MS04], we describe the three components embedded to the SDDS2000 architecture. We report also performance results related to parity overhead, data recovery (server and record).

[LMS04] is a demo paper, where we summarize our LH*$_{RS}$ best settings, describing at a glance our RS encoding/decoding scheme as well as our server architecture. The demo outline shows the main functionalities of our prototype.

Finally, in [LMS04-*b*], we detail LH*$_{RS}$ fundamental theory with the new parity matrix (column and line of '1's). We highlight our encoding/decoding optimizations and report performance results. We also discuss the related work. This paper is submitted for a journal publication

Thesis Outline

The Thesis dissertation is divided into two parts, plus appendixes. The first part, *Etat de l'Art*, covers the state of the art of the networked data storage systems, and mechanisms of high availability. It consists of two chapters. The second part, *Conception & Implantation de LH*$_{RS}$*, describes LH*$_{RS}$ schema and our LH*$_{RS}$ manager as well as performance results.

The first Chapter of the Thesis, "*Introduction* ", discusses the basic issues and the motivation for the Thesis work, as well as our contribution.

In Chapter 2, "*Introduction aux Systèmes Répartis* ", we cover the basic features of networked data storage systems: hardware configuration, data distribution schemes, server architectural design, communication protocols, evaluation metrics and

requirements. We also cover with special interest Scalable and Distributed Data Structures (SDDS), being one of our thesis fundamentals.

Chapter 3, "*Revue de Mécanismes de Haute Disponibilité*" reviews literature on high availability. It sketches recovery techniques from media failures [PGK88, HGK+94, B3M95, XB99, SS96, SS02, CEG+04, LMR98]. We explore and highlight in turn the pros and cons of each of these techniques.

Chapter 4, "*Fondements Théoriques de LH*$^*_{RS}$", details Reed Solomon codes, LH*$_{RS}$ schema, gives some hints to improve coding and decoding, and compares our adaptation of Reed Solomon codes to related work [R89, W91, SB92, BM93, WB94, BKL+95, SB95, R96, R97, P97, ABC97, IMT03, MTS04].

Chapter 5, "*Le Gestionnaire LH*$^*_{RS}$", describes the three components embedded to SDDS2000 server architecture, namely the TCP/IP connection handler [ISI81, MB00], the message acknowledgement strategy [GRS97, D01] and the dynamic addressing structure.

Chapter 6, "*Architecture opérationnelle du Gestionnaire LH*$^*_{RS}$", describes different scenarios, especially client manipulation (insert, update, delete, search queries), update parity propagation during the server split, parity server creation and servers recovery. We also discuss how the scenarios are mapped to the different architectures.

Chapter 7, "*Mesures de Performances de LH*$^*_{RS}$", reports the results of the conducted experiments in two different hardware configurations: a set of five 733MHz machines connected to a 100Mbps Ethernet network and a set of five 1.8GHz machines connected to 1Gbps Ethernet network. The performance factors of interest are the insert record time in a *k*-available LH*$_{RS}$ scheme, the record search time in both normal mode and degraded mode, the parity bucket creation time and *f* buckets' recovery time. We compare performance results obtained in the two different configurations, SDDS2000 server architecture to our enhanced server architecture, and RS/ XOR coding/decoding in the two Galois Fields GF(2^8) and GF(2^{16}).

Finally, the last chapter "*Conclusion & Travaux Futurs*", concludes the thesis dissertation and gives some future research directions.

The Thesis contains also *Appendix A & B* that present our prototype. We describe there the main functions offered by our client interface. We also describe the installation requirements, the structure of our object code for future extensions.

Table des Matières

Liste des Figures .... **xviii**
Liste des Scénarios .... **xx**
Liste des Algorithmes .... **xxi**
Liste des Tableaux .... **xxii**
1 **Introduction** **1**
1.1 Progrès Technologique 1
1.2 Les Structures de Données Distribuées & Scalables 2
1.3 Problématique de la Haute Disponibilité 3
1.4 Contribution de la Thèse 5
1.5 Plan de la Thèse 6

PARTIE I : ETAT DE L'ART **8**
2 **Introduction aux Systèmes Répartis** **9**
2.1 Introduction 9
2.2 Architectures Parallèles 9
2.2.1 Architecture à Mémoires Partagées 9
2.2.2 Architecture à Disques Partagés 10
2.2.3 Architecture à Mémoires Distribuées 10
2.2.4 Architectures hybrides 10
2.3 Distribution de Données & Traitement Parallèle 11
2.3.1 Stratégies de Distribution de Données 12
2.3.2 Stratégies de Traitement Parallèle 12
2.4 Architectures de Systèmes 13
2.4.1 Architecture Globale 13
2.4.2 Architectures de Serveurs 14
2.5 La Communication Réseau 17
2.5.1 Protocole UDP 18
2.5.2 Protocole TCP 18
2.6 Les Structures de Données Distribuées et Scalables 19
2.6.1 Distribution par Hachage 19
2.6.2 Distribution par Intervalle 24
2.7 Critères d'évaluation d'un Système Réparti 25
2.7.1 Les 12 Critères de Conception d'un Système Réparti de Date 26
2.7.2 Facteur de Rapidité 26

2.7.3 Facteur d'Echelle 28
2.7.4 Loi d'Amdhal 29
2.8 Conclusion 29
3 Revue de Mécanismes de Haute Disponibilité 30
3.1 Introduction 30
3.2 La Haute Disponibilité 30
3.2.1 Classification de Systèmes à Haute Disponibilité 30
3.2.2 Fiabilité d'un Système 32
3.2.3 Critères d'Evaluation d'une Solution 32
3.2.4 Codes de Correction d'Effacements versus Réplication 33
3.3 Réplication 33
3.3.1 Schéma de Réplication *SSS* 33
3.3.2 Schémas de Réplication Optimisés 35
3.4 La Technologie RAID 36
3.5 Codes Linéaires Binaires 38
3.5.1 Codage/ Décodage 38
3.5.2 Le code *1d-parity* 39
3.5.3 Les Codes Correcteurs à 2 Effacements 39
3.5.4 Les Codes Correcteurs de 3 Effacements 41
3.5.5 Conclusion 42
3.6 Schéma EVENODD 42
3.6.1 Processus de Codage 43
3.6.2 Processus de Décodage 44
3.6.3 Discussion 45
3.7 Schéma X-Code 45
3.7.1 Processus de Codage 46
3.7.2 Processus de Décodage 47
3.7.3 Conclusion 52
3.8 Schéma *Schwabe-Sutherland* 52
3.9 Schéma *Row-Diagonal Parity* 54
3.10 Les SDDSs à Haute Disponibilité 55
3.10.1 LH^*_M 56
3.10.2 LH^*_S 56
3.10.3 Adaptation du concept de groupage à LH* 57
3.11 Conclusion 58

PARTIE II : LH^*_{RS}, UNE SDDS A HAUTE DISPONIBILITE 60
4 Fondements Théoriques de LH^*_{RS} 61

4.1 Introduction 61
4.2 Les Codes Reed Solomon 61
4.2.1 Principe 61
4.2.2 Les Corps de Galois 62
4.2.3 La Matrice Génératrice 64
4.2.4 Code Systématique 65
4.2.5 Codage RS 66
4.2.6 Décodage RS 67
4.3 Le schéma de Haute disponibilité LH^*_{RS} 68
4.3.1 Le Fichier LH^*_{RS} 69
4.3.2 Evolution d'un Fichier LH^*_{RS} 70
4.3.3 Optimisation du Codage/ Décodage RS 72
4.3.4 Calcul de Parité dans LH^*_{RS} 74
4.3.5 Récupération dans LH^*_{RS} 80
4.3.6 Gestion des Rangs dans LH^*_{RS} 82
4.4 Schémas de Haute Disponibilité à base des Codes Reed Solomon 83
4.4.1 Travaux de Blömer et al. 84
4.4.2 Projet Koh-i-Noor de Microsoft 86
4.5 Conclusion 87
5 Le Gestionnaire LH^*_{RS} 88
5.1 Introduction 88
5.2 Architecture SDDS2000 89
5.2.1 Client SDDS2000 89
5.2.2 Architecture d'une case SDDS2000 90
5.2.3 Les connexions TCP/IP dans SDDS2000 91
5.3 Architecture SDDS-TCP 92
5.4 Architecture SDDS-Ack 94
5.4.1 Stratégie d'Acquittement dans SDDS2000 95
5.4.2 Le Protocole TRAP 95
5.4.3 Stratégie d'Acquittement Proposée 96
5.4.4 CFAM au Niveau du Client 97
5.4.5 CFAM au Niveau d'une Case de Données 99
5.4.6 Exemple 102
5.5 Architecture SDDS-IP@ 104
5.5.1 Structures d'adressage 104
5.5.2 Groupes Multicast 107
5.5.3 Architecture 108
5.6 Architecture Fonctionnelle 109

5.7 Conclusion 110
6 Architecture Opérationnelle du Gestionnaire LH*$_{RS}$ 111
6.1 Introduction 111
6.2 Scénarios de Manipulations Client 113
6.2.1 Insertion d'un Enregistrement 113
6.2.2 Suppression d'un Enregistrement 114
6.2.3 Mise à Jour d'un Enregistrement 115
6.3 Scénario d'Eclatement d'une Case de Données LH*$_{RS}$ 117
6.3.1 Eclatement d'une Case de Données LH*$_{LH}$ 117
6.3.2 Transfert des Mises à Jour par UDP 118
6.3.3 Transfert des Mises à Jour par TCP 119
6.3.4 Impact de l'Architecture de Cases sur le Scénario d'Eclatement 122
6.3.5 Discussion de Variantes 122
6.4 Scénario d'ajout d'une Case de Parité à un Groupe 124
6.4.1 Création d'une Case de Parité par UDP 124
6.4.2 Création d'une Case de Parité par TCP/IP 126
6.4.3 Scénario Création Case de Parité vs Architecture SDDS 127
6.4.4 Discussion 128
6.5 Scénario de Récupération 130
6.5.1 Détecteur de pannes 130
6.5.2 Diagnostic d'un Groupe de Parité 131
6.5.3 Procédures de Décodage 133
6.5.4 Récupération par UDP 135
6.5.5 Récupération par TCP 135
6.6 Conclusion 140
7 Mesures de Performances de LH*$_{RS}$ 141
7.1 Introduction 141
7.2 Environnement Expérimental 141
7.2.1 Configuration Matérielle 141
7.2.2 Paramètres de LH*$_{RS}$ 142
7.3 Optimisation du Calcul de Parité 143
7.3.1 Pré-calcul du *log* 143
7.3.2 Nouvelle Matrice Génératrice 143
7.4 Création d'un fichier 144
7.4.1 Architecture SDDS2000 145
7.4.2 Architecture SDDS-TCP 146
7.4.3 Architecture SDDS-Ack 148
7.4.4 Conclusion 149

7.5 Requêtes de Recherche 149
7.5.1 Première Configuration 150
7.5.2 Deuxième Configuration 150
7.5.3 Conclusion 151
7.6 Récupération d'enregistrements de données 151
7.6.1 Description des expérimentations 151
7.6.2 Première Configuration 153
7.6.3 Deuxième Configuration 154
7.6.4 Comparaison 154
7.7 Requêtes de Mise à Jour Aveugles 155
7.7.1 Description des Expérimentations 155
7.7.2 Performances de MAJs 155
7.8 Création d'une Case de Parité 156
7.8.1 Description des Expérimentations 156
7.8.2 Architecture des Cases SDDS2000 157
7.8.3 Architecture des Cases SDDS-TCP 157
7.8.4 Comparaison des 2 Architectures 159
7.9 Récupération de Cases de Données 160
7.9.1 Description des Expérimentations 160
7.9.2 Architecture de Cases SDDS2000 160
7.9.3 Architecture de Cases SDDS-TCP 163
7.10 Conclusion 168
8 Conclusion & Travaux Futurs 169
8.1 Introduction 169
8.2 Mise en Œuvre plus Performante 169
8.2.1 Récupération de Cases 169
8.2.2 Récupération d'Enregistrements de Données 170
8.2.3 Transfert des Mises à Jour vers les Cases de Parité 170
8.2.4 Eclatement d'une Case de Données 171
8.2.5 La Structure de Données d'une Case 171
8.3 Perspectives d'un Nouveau Schéma LH^*_{RS} 171
8.3.1 Le *coordinateur* 171
8.3.2 Schéma d'Allocation 171
8.3.3 « Parity Declustering » 172
8.4 Expérimentation de Techniques Alternatives de Codage/Décodage 172
8.5 Communication plus Rapide 172
8.5.1 Communication par UDP 172
8.5.2 I/O Completion Ports 173

8.5.3 Le Protocole T/TCP ... 173
8.6 Conclusion ... 174
Bibliographie ... 175
Annexe A: Description du Prototype ... 184
Annexe B: Prototype User Guide ... 186

PUBLICATIONS ... 189

LISTE DES FIGURES

Figure 1-1 : Loi de Gilder versus Loi de Moore dans les 20 prochaines années [A01]. 2
Figure 2-1: Architectures Matérielles Parallèles. 11
Figure 2-2 : Stratégies de Distribution des Données. 12
Figure 2-3 : Stratégies de Traitement Parallèle. 13
Figure 2-4 : Architecture Client Serveur à 2 Strates.. 14
Figure 2-5 : Classification des Architectures selon Welch et al. [WGBC00]. 16
Figure 2-6 : Allure des courbes de performances des architectures multi-tâches et des architectures orientées-événement. 17
Figure 2-7 : Allure des courbes de performances d'une architecture hybride. 17
Figure 2-8 : Position des *Sockets* dans le modèle OSI... 18
Figure 2-9 : Illustration de la SDDS LH*.. 20
Figure 2-10 : Structure d'une Case LH^*_{LH}. 23
Figure 2-11 : Exemple d'évolution d'un fichier RP*. 25
Figure 2-12 : Illustration du Facteur de Rapidité... 27
Figure 2-13: Courbe idéale du Facteur de Rapidité. 27
Figure 2-14 : Illustration du Facteur d'Echelle.. 28
Figure 2-15 : Courbe idéale du Facteur d'Echelle. 28
Figure 3-1 : Eclatement de deux nœuds *u* et *u'*, et création d'un nouveau serveur *S3*...... 34
Figure 3-2: Schéma *Chained-Declustering* pour $m = 8$. 35
Figure 3-3: Schéma Multi-Chained-Declustering pour $m = 8$. 36
Figure 3-4 : RAID niveau 5. 37
Figure 3-5 : Illustration du schéma *1d-parity* pour ($k = 16$, $c = 4$). 39
Figure 3-6 : Illustration du *2d-parity* pour ($k = 16$, $c = 4$). 40
Figure 3-7 : EVENODD, pour $m = 5$... 44
Figure 3-8 : Illustration de X-Code ($m = 5$)... 46
Figure 3-9 : Lignes de parité du X-code. 47
Figure 3-10 : Equations de Codage du Schéma X-code, $m = 5$, perte de la colonne 2. 48
Figure 3-11 : Equations de Codage du Schéma X-code, $m = 5$, perte des colonnes 1 et 3. 49
Figure 3-12: Schéma *mod 0* ; $R = 6$, $D = 3$ et $t = 3$. 53
Figure 3-13: Illustration du schéma Row-Diagonal Parity pour p = 5. 55
Figure 3-14: Aperçu de différentes SDDSs. 56
Figure 3-15: Insertion d'un enregistrement de clef 53 dans un fichier LH^*_S ($k = 3$). 57
Figure 4-1: Illustration des processus de codage et décodage RS. 62
Figure 4-2: Représentation de la matrice de Vandermonde. 64
Figure 4-3: Représentation de la matrice de Cauchy. 65
Figure 4-4: Processus de codage.. 67
Figure 4-5: Un groupe de parité d'un fichier LH^*_{RS} ($m = 4$, $k = 2$). 69
Figure 4-6: Structure des enregistrements du fichier LH^*_{RS}. 70
Figure 4-7: Evolution d'un fichier LH^*_{RS}. 71
Figure 4-8: Exemple d'une matrice de Vandermonde transformée, cas $CG(2^8)$ et $m = 4$. 73
Figure 4-9: Matrices P (4×3) et Q (4×3), pour ($m = 4$, $CG(2^8)$). 76

Figure 4-10: Exemples d'insertions en ligne. 77
Figure 4-11: Exemple de suppression d'un enregistrement. 78
Figure 4-12: Exemples de Mises à Jour en ligne. 80
Figure 4-13 : Exemple de récupération d'un enregistrement par décodage XOR. 81
Figure 5-1: Architecture d'un Client SDDS2000. 90
Figure 5-2: Architecture d'une case SDDS2000. 91
Figure 5-3: Architecture d'une case de données ou de parité dans SDDS-TCP. 94
Figure 5-4: Structure de *Liste Messages Non Encore Acquittés* 97
Figure 5-5: Architecture d'un client SDDS-Ack. 99
Figure 5-6: Structures d'acquittement niveau case de données. 100
Figure 5-7: Architecture d'une Case SDDS-Ack. 101
Figure 5-8: Exemple de déroulement de l'algorithme d'acquittement. 104
Figure 5-9: Structures d'adressage du Coordinateur. 106
Figure 5-10: Structures d'adressage d'une case de données. 106
Figure 5-11: Structure d'adressage d'une Case de Parité. 107
Figure 5-12: Structure d'adressage d'un client. 107
Figure 5-13: Architecture SDDS-IP@ d'une case. 109
Figure 5-14: Architecture Fonctionnelle d'une Case de Données. 109
Figure 5-15: Architecture Fonctionnelle d'une Case de Parité. 110
Figure 6-1: Architecture globale de SDDS-2000. 112
Figure 7-1: Performances de création d'un fichier d'un fichier LH^*_{RS} k-disponible -[k =0, 1, 2 ; Architecture SDDS2000, 1ére config.] 145
Figure 7-2: Temps moyen d'insertion d'un enregistrement dans un LH^*_{RS} k-disponible -[k = 0, 1, 2 ; Architecture SDDS2000, 1ére config.]. 146
Figure 7-3: Performances de création d'un fichier LH^*_{RS} k-disponible -[k = 0, 1, 2 ; Architecture SDDS-TCP ; 1ére config.]. 147
Figure 7-4: Performances de création d'un fichier LH^*_{RS} k-disponible -[k = 0, 1, 2 ; Architecture SDDS-TCP ; 2éme config.]. 147
Figure 7-5: Performances de création d'un fichier LH^*_{RS} k-disponible -[k = 0, 1, 2 ; Architecture SDDS-Ack ; 2éme config.]. 149
Figure 7-6: Performances des requêtes de recherche dans la 1ére configuration. 150
Figure 7-7: Performances des requêtes de recherche dans la 2éme configuration. 150
Figure 7-8: Récupération de x enregistrements de données dans la première configuration, par décodage [RS, CG[2^{16}]], [RS, CG[2^8]], [XOR, CG[2^{16}]],[RS, CG[2^8]]. 153
Figure 7-9: Récupération d'enregistrements de données dans la deuxième configuration, par décodage [RS, CG[2^{16}]], [RS, CG[2^8]], [XOR, CG[2^{16}]],[RS, CG[2^8]]. 154

LISTE DES SCENARIOS

Scénario 5-1: Etablissement d'une Connexion TCP/IP dans l'architecture SDDS2000... 92
Scénario 5-2: Etablissement d'une Connexion TCP dans l'architecture SDDS-TCP....... 93
Scénario 6-1: Insertion d'un Enregistrement de Données. 113
Scénario 6-2: Suppression d'un enregistrement de données. 115
Scénario 6-3: Mise à jour d'un enregistrement de données.. 116
Scénario 6-4: Eclatement d'une case $LH*_{LH}$. 118
Scénario 6-5: Eclatement d'une case de données $LH*_{RS}$ [L00] 119
Scénario 6-6: Eclatement d'une case de données $LH*_{RS}$ proposé................................. 121
Scénario 6-7: Ajout d'une case de données dans l'architecture SDDS-IP@. 123
Scénario 6-8: Attacher *k* cases de parité à un groupe de parité dans l'architecture SDDS-IP@. 124
Scénario 6-9: Ajout d'une Case de Parité [Coordinateur – Case de Parité]. 125
Scénario 6-10: Ajout d'une Case de Parité par UDP... 126
Scénario 6-11: Ajout d'une Case de Parité par TCP/IP.. 127
Scénario 6-12: Ajout d'une case de parité dans l'architecture SDDS-IP@. 129
Scénario 6-13: Diagnostic d'un groupe de parité. 132
Scénario 6-14: Récupération de l'ensemble des cases de parité par UDP. 136
Scénario 6-15: Récupération mixte par UDP. 137
Scénario 6-16: Récupération par TCP/IP et par Tranche d'enregistrements.................. 138
Scénario 6-17: Choix des cases de secours dans l'architecture SDDS-IP@. 139

LISTE DES ALGORITHMES

Algorithme 2-1 : Adressage logique LH d'un enregistrement de clé *c* 19
Algorithme 2-2 : Mise à jour du couple (*i*, *n*) après un éclatement d'une case. 20
Algorithme 2-3 : Mise à jour du couple (*i*, *n*) après la fusion de deux cases. 20
Algorithme 2-4 : Adressage exécuté par le client.. 21
Algorithme 2-5 : Ajustement de l'image client. 21
Algorithme 2-6 : Test & Renvoi exécuté par la case... 21
Algorithme 3-1: Codage dans le schéma EVENODD... 43
Algorithme 3-2 : Décodage dans le schéma EVENODD. 44
Algorithme 3-3 : Equations calculant les lignes *m*-1 et *m*. 46
Algorithme 3-4 : Récupération d'une colonne dans *X-code*.. 48
Algorithme 3-5 : Récupération de 2 colonnes dans *X-code*. 49
Algorithme 3-6 : Résoudre sachant un symbole, décodage X-code................................. 51
Algorithme 3-7: Calcul des unités du disque de parité P_j dans le schéma de Sutherland. 53
Algorithme 3-8: Equations de codage RDP –calcul des colonnes *p-1* et *p*. 54
Algorithme 4-1: Diagonalisation d'une Matrice G.. 66
Algorithme 4-2: Codage RS. 67
Algorithme 4-3: Décodage RS.. 68
Algorithme 4-4: Multiplication de Ed_j par ξ 85
Algorithme 5-1: Re-émission de messages niveau client. 98
Algorithme 5-2 : Re-émission de messages niveau case de données. 101
Algorithme 6-1: Insertion d'un enreg. de données par une case de parité. 114
Algorithme 6-2: Suppression d'un enreg. de données par une case de parité. 115
Algorithme 6-3: Mise à jour d'un enregistrement de données par une case de parité..... 116
Algorithme 6-4: Mise à jour optimisée d'un enregistrement de données par une case de parité. 117
Algorithme 6-5: Procédure de Traitement d'une structure S du tampon de mises à jour. 121
Algorithme 6-6: Test de Récupération. 132
Algorithme 6-7: Décodage RS [L00] 133
Algorithme 6-8: Décodage RS optimisée [M00].. 133
Algorithme 6-9: Décodage XOR.. 134
Algorithme 6-10: Récupération RS d'enregistrements de parité.................................... 134
Algorithme 6-11: Récupération XOR d'enregistrements de parité. 134
Algorithme 6-12: Décodage RS en ligne... 135
Algorithme 6-13: Décodage XOR en ligne. 135
Algorithme 8-1: Recherche d'un enregistrement de données de clef *C* dans une case de parité. 170

LISTE DES TABLEAUX

Table 1-1: Coût moyen d'une heure d'indisponibilité pour différentes applications [CPR]. 3
Table 3-1: Classification de systèmes en fonction du temps d'indisponibilité par an [GS91]. 31
Table 3-2 : Comparaison des codes *2d-parity* et *full-2* 40
Table 3-3 : Comparaison des codes *3d-parity, full-3, Steiner* et *additive-3*. 42
Table 4-1: Représentation polynomiale des éléments de CG(2^3)\{0}, $p(x) = x^3+x+1$ 63
Table 4-2: Représentation des éléments de CG(2^3). 63
Table 4-3: Tables *log* et *antilog* pour CG(2^3). 64
Table 4-4: La table *log* de CG(2^8). 75
Table 4-5 : Exemple de récupération d'enregistrements par décodage RS. 82
Table 7-1: Comparaison des temps de traitement de création d'une case de parité (en ms) dans la 2éme config. 143
Table 7-2: Comparaison des temps de traitement de création d'une case de parité (en ms), dans la 2éme config. 144
Table 7-3: Temps d'insertion d'un enregistrement (ms) dans un LH*$_{RS}$ *k*-disponible -[k = 0, 1, 2 ; Architecture SDDS-TCP ; 2éme config.]. 148
Table 7-4: Analyse des performances de requêtes parallèles ciblant un seul serveur de données dans la 2éme Configuration. 151
Table 7-5: Temps de mise à jour d'un enregistrement de données et de *k* enregistrements de parité, dans un fichier LH*$_{RS}$, *k*-disponible, dans CG[2^8]. 155
Table 7-6: Temps de mise à jour d'un enregistrement de données et de *k* enregistrements de parité, dans un fichier LH*$_{RS}$, *k*-disponible, dans CG[2^{16}]. 156
Table 7-7: Temps de création d'une case de parité, Architecture SDDS2000, codage RS dans CG[2^8]. 157
Table 7-8: Performances de création d'une case de parité, pour différentes tailles de cases, et différents algorithmes de codage, dans la 1ére config. 158
Table 7-9: Performances de création d'une case de parité, pour différentes tailles de cases, et différents algorithmes de codage, dans la 2éme config. 159
Table 7-10: Temps d'inversion d'une matrice $m \times m$ en µs. 160
Table 7-11: Performances de récupération de f cases de données, f = 1, 2, 3 -[Taille récupérée f*3,125MO ; UDP ; décodage RS ; CG[2^8]; SDDS2000 ; 1ére config.]. 161
Table 7-12: Performances de récupération de f cases de données, f = 1, 2, 3, [Taille récupérée f*3,125MO ; TCP/IP ; décodage RS ; CG[2^8] ; SDDS2000 ; 1ére config.]. 162
Table 7-13: Variation du temps de récupération en fonction du paramètre *Tranche*. 163
Table 7-14: Performances de récupération d'une case de données. [Taille récupérée : 3,125MO ; TCP/IP ; SDDS-TCP; 1ére config.]. 164
Table 7-15: Récupération d'1 case de données –[Taille récupérée 3,125MO ; TCP ; SDDS-TCP; 2éme config.]. 165

Table 7-16: Récupération de 2 cases de données -[Taille récupérée 6,250MO ; TCP ; décodage RS ; 1ére config.].. 165
Table 7-17: Performances de récupération de 2 cases de données -[Taille récupérée 6,250MO ; TCP ; Architecture SDDS-TCP; 2éme config.]. 166
Table 7-18: Performances de récupération de 3 cases de Données -[Taille récupérée 9,325MO ; TCP ; décodage RS ; SDDS-TCP; 1ére config.].................................. 167
Table 7-19: Performances de récupération de 3 cases de Données -[Taille récupérée 9,325MO ; TCP ; SDDS-TCP; 2ére config.].. 167

CHAPITRE

1 INTRODUCTION

1.1 Progrès Technologique

Le succès de la nouvelle économie est incontestablement dû à l'augmentation des capacités informatiques et la diminution de leur prix. Plusieurs recherches se sont intéressées à la définition de règles pour le dimensionnement des systèmes, et quelques lois ont été proposées. Parmi, les plus connues, sont la loi de Moore et la loi de Gilder.

La Loi de Moore

Gordon E. Moore [M65, M03], un des fondateurs d'Intel, a établi une 'loi' à partir des deux observations suivantes:

* Chaque nouveau circuit contient deux fois plus de transistors que sa version précédente,
* Une nouvelle génération de microprocesseurs est lancée en moyenne tous les dix-huit mois.

La loi de Moore a été énoncée au début de l'industrie de la microélectronique de façon empirique et s'est vérifiée au cours des années. Elle spécule que les vitesses des processeurs et les capacités des volumes de stockage (mémoires secondaires, mémoires centrales...) doublent tous les dix huit mois. D'ailleurs, en novembre 2001, Intel a annoncé la fabrication dès 2007 de puces *Térahertz* intégrant un milliard de transistors et fonctionnant à 1000 Ghz.

La Loi de Gilder

Comme dans le domaine des microprocesseurs avec la loi de Moore, la loi de Gilder spécule l'évolution des réseaux de communication. Ainsi, en 1995, Georges Gilder prédit pour les vingt cinq prochaines années, que la bande passante des réseaux triplerait à prix égal tous les ans et que le débit des liaisons quadruplerait tous les trois ans [G97].

Loi de Gilder versus Loi de Moore

La Figure 1-1 illustre sur la même échelle de temps, l'évolution des vitesses des processeurs –mesurée en Gflop/ sec, de la bande passante des réseaux de communication –mesurée en Gbps et des capacités de stockage –mesurée en Goctets.

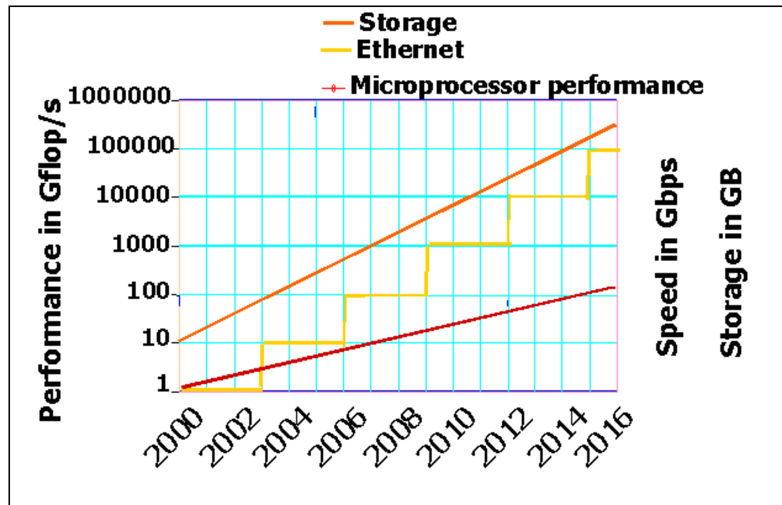


Figure 1-1 : Loi de Gilder versus Loi de Moore dans les 20 prochaines années [A01].

Malgré les critiques dont font l'objet ces deux lois, un progrès confirmé continuera à se réaliser et affectera en premier plan l'architecture des ordinateurs et l'infrastructure réseau, et en deuxième plan la conception des applications informatiques.

1.2 Les Structures de Données Distribuées & Scalables

Certes le progrès technologique est indéniable, il touche les microprocesseurs -dont la vitesse augmente de 50% par an, et les capacités de stockages -à la fois disques et DRAM quadruplent de taille tous les trois ans. Cependant, ce progrès reste modeste en ce qui concerne le débit des disques, qui sur les dix dernières années, a juste doublé. Les experts parlent de performance absolue des processeurs et de différentiel processeur-mémoire-disque. Il y'a lieu de noter également que face au progrès technologique, le volume d'information augmente de 30% par an. Ces conjonctions sont la cause du goulot d'étranglement sur les E/Ss, et ont poussé les chercheurs à transiger avec les systèmes de stockage de données centralisés, pour proposer des systèmes de stockage de données distribués. En effet, la distribution des données sur plusieurs nœuds de stockage augmente le débit des E/Ss en procurant de meilleurs temps d'accès aux données.

Dans cette direction, plusieurs laboratoires de recherche et éditeurs de Systèmes de Gestion de Bases de Données (SGBD) ont développé des SGF et SGBD distribués (NFS [SGL+85], AFS [HKM+88], Petal [LT96],Oracle [DA99], Ms SQL Server 2K, …) et ont ainsi concrétisé les principes d'exploitation des capacités de stockage existantes, de distribution de données et de traitement parallèle des opérations. Toutefois, les prototypes de recherche et les logiciels sont encore loin des 12 critères de conception d'un système réparti établis par C. J. Date [D87].

Les Structures de Données Distribuées et Scalables (SDDS) se proposent de distribuer les données sur plusieurs nœuds de stockage. Le fichier de données, ainsi créé, (1) augmente de taille de manière dynamique par l'éclatement de serveurs suite à la surcharge de serveurs et (2) se rétrécit par fusion de serveurs suite aux suppressions d'articles. Les

SDDSs sont proposées pour fournir à haut débit, un bon temps de réponse, sur des gros volumes de données.

Malgré les différents avantages que procure la distribution des données sur plusieurs nœuds, la vulnérabilité aux pannes est un problème qui s'accentue avec l'augmentation du nombre de machines dans le réseau. Cette thèse s'intéresse au problème de haute disponibilité dans les SDDSs.

1.3 Problématique de la Haute Disponibilité

Nombreuses sont les entreprises qui ne tolèrent pas la perte de leurs données ou que leurs systèmes informatiques s'arrêtent. Certaines ont un système informatique contenant des Téraoctets de données qui ne peuvent pas être perdues sans conséquences financières catastrophiques. A cet effet, les applications de télécommunication, aéroportuaires, médicales ou bancaires, ne peuvent tolérer une interruption de service. Toute interruption de service aura pour conséquence une réduction de productivité, une baisse du chiffre d'affaires, la notoriété de l'entreprise en question, des dégâts matériels et dans certains cas humains.

Plusieurs sources peuvent nous renseigner sur le coût d'indisponibilité d'un service dans un domaine déterminé. Les chiffres dépendant de la méthode de quantification des pertes peuvent diverger, mais convergent sur l'importance de la haute disponibilité.
La Table 1-1 (de source des statistiques faites en 1996 par le *Contingency Planning Research*) donne quelques chiffres illustratifs du coût moyen d'une heure d'indisponibilité dans différents domaines.

Application	*Secteur d'activité*	*Coût de l'indisponibilité*
Courtage	Finance	$6,45 millions
Vente par carte de crédit	Finance	$2.6 millions
Ebay	Vente	$225 000
Films à la demande	Loisirs	$150 000
Téléachat	Distribution	$113 000
Ventes sur catalogue	Distribution	$90 000
Réservation aérienne	Transport	$89 500

Table 1-1: Coût moyen d'une heure d'indisponibilité pour différentes applications [CPR].

Les échecs sont de plusieurs types, nous distinguons (*1*) des échecs de type logiciel où l'application présente des défaillances ; (*2*) des échecs dus à une faute humaine en général il s'agit d'erreurs de configuration ou de mauvaise manipulation ; (*3*) et des échecs de type matériel résultant d'échec de sites ou de liens de communication. Ces derniers sont dus à des catastrophes naturelles ou de sabotage (tremblement de terre, inondation, incendie …) comme ils peuvent également être dus à la fiabilité du matériel informatique. Dans cette thèse, on s'intéresse à la récupération des données après échec d'un nœud de stockage dans un système de stockage de données distribué.

Plusieurs chercheurs ont manifesté un intérêt particulier à l'égard des systèmes de stockage distribués à haute disponibilité et différentes solutions sont proposées. Certaines solutions proposées se sont avérées prohibitives, en terme de stockage ou de complexité de récupération. Notons qu'une stratégie de haute disponibilité est évaluée par respect aux critères suivants et à titre égal:

(1) Le coût de déploiement de la solution (stockage, temps CPU),

(2) Le temps de reprise après échec,

(3) La résistance aux échecs,

(4) La dégradation des performances en cas d'échec.

La solution la plus triviale de haute disponibilité consiste à sauvegarder les données régulièrement sur des supports magnétiques ou optiques. En cas de perte, les données sont récupérées à partir des sauvegardes, avec le risque de perte des mises à jour ayant lieu après la dernière sauvegarde. Le principe des miroirs ou haute disponibilité par réplication, découle du principe de sauvegarde. Il consiste à sauver les données sur d'autres nœuds de stockage et à définir une stratégie de propagation des mises à jour (synchrone ou asynchrone et incrémentale ou écrasement) vers les sites miroirs. L'atout majeur est que les sites miroirs sont fonctionnels et interrogeables, par contre le principal inconvénient des solutions à base de réplication de données est le coût de stockage qui est un facteur du nombre de répliquas.

Des solutions plus économiques de point stockage consistent en le calcul de parité. Le schéma de calcul de parité le plus simple est celui se basant sur la parité impaire (ou-exclusif, XOR, ^, ⊕). Pour un groupe de nœuds de stockage de données, par exemple N_x, N_y, N_z ; la parité impaire est définie par l'équation de codage suivante :

$$\text{Données de } N_p = \text{Données de } N_x \wedge \text{ Données de } N_y \wedge \text{ Données de } N_z$$

Le schéma est dit 1-disponible, car il tolère seulement l'échec d'un nœud dans un groupe de nœuds. En effet, en cas d'échec du nœud N_x, les données sont simplement récupérées en utilisant les données de tout le groupe, càd de N_y, N_z et N_p et l'équation de décodage suivante :

$$\text{Données de } N_x = \text{Données de } N_p \wedge \text{ Données de } N_y \wedge \text{ Données de } N_z$$

D'autres schémas à haute disponibilité cherchant la k-disponibilité ($k > 1$) ont fait l'objet d'investigations. Ces schémas trouvant leurs fondements dans la théorie de l'information, exploitent des techniques de codage pour le calcul des données de parité et de décodage pour la récupération. Nous distinguons les *arrays codes* dont le calcul de parité se base sur le ou-exclusif (XOR, ^, ⊕) [B3M95, XB99, CEG+04], et les codes Reed-Solomon [RS63], dont le calcul de parité s'exécute dans le corps de Galois et est plus complexe. Les codes Reed Solomon permettent de tolérer un grand nombre d'échecs de volumes de stockage, et ce par le calcul de plus de données de parité [R89, W91, SB92, BM93, WB94, BKL+95, SB95, R96, R97, P97, ABC97, LS00, IMT03, MTS04].

Notons que, dans le domaine, certains prototypes de recherche se sont concrétisées notamment la technologie RAID (acronyme de *Redundant Arrays of Inexpensive Disks*). Cette dernière a été proposée en 1987 par trois chercheurs de l'université de Berkeley D.A. Patterson, G. Gibson & R.H. Katz [PGK88]. La technologie RAID offre plusieurs

niveaux où les solutions de haute disponibilité réplication et/ou parité sont implantées. A présent, la technologie RAID est largement répandue sur le marché informatique comme architecture de stockage à haut débit et à haute disponibilité.

1.4 Contribution de la Thèse

Dans le cadre de cette thèse, notre objectif principal est l'étude, la conception et la mise en œuvre de la SDDS à haute disponibilité LH^*_{RS}, et la proposition d'un gestionnaire LH^*_{RS}. La Structure de Données Distribuée et Scalable LH^*_{RS} [LS00] résout le problème de la tolérance aux pannes, particulièrement l'échec de nœuds par l'ajout de données redondantes calculées par les codes Reed Solomon.

Dans une première étape, nous avons défini les mots clefs du thème de recherche afin de prendre connaissance des travaux de recherche s'intéressant au sujet. Nos recherches sont au confluent des structures de données distribuées, des architectures des serveurs, des mécanismes de haute disponibilité et de la théorie de l'information et du codage.

Tout au long de la thèse, nous avons réalisé un cycle itératif de conception, mise au point puis validation expérimentale. Nous avions pour but la recherche de meilleurs résultats. Ces derniers touchent séparément la *Composante Réseau*, mesurée par le *Temps de Communication* et la *Composante CPU*, mesurée par le *Temps de Traitement*. La *Composante Réseau* a été améliorée par la proposition de nouvelles architectures pour les serveurs, notamment (*1*) un gestionnaire de connections TCP/IP proposé sur la base de la RFC 793 [ISI81] et l'implantation de TCP/IP sous Windows [MB00], (*2*) une stratégie de contrôle de flux et d'acquittement de messages sur la base des travaux [J88,GRS97, D01] et (*3*) une table d'adresses dynamique qui évolue avec le fichier, tel que les paramètres de communication des serveurs sont déterminés par le coordinateur. Les nouveaux serveurs sont contactés par Multicast. Les trois composants sont intégrés à l'architecture de client/ serveur SDDS2000 de LH^*_{RS} [B00, D01].

Quant à la *Composante CPU*, elle a été améliorée par la recherche d'optimisations pour les processus codage/décodage. La première optimisation concerne la proposition d'une nouvelle matrice de parité qui réduit la complexité du codage et du décodage. La nouvelle matrice a une colonne de '1's, permettant de coder les données de parité du premier serveur de parité par calcul XOR, et permettant également un décodage XOR seulement en cas d'échec d'un serveur de données et de disponibilité du premier serveur de parité. La nouvelle matrice a aussi une ligne de '1's, qui permet de réduire les mises à jour parvenant du premier serveur de données en simple calcul XOR. Nous avons également amélioré le codage et décodage RS, par le pré-calcul du *log*, ce qui réduit la complexité des multiplications dans le corps de Galois.

Pour la maintenance de la SDDS à haute disponibilité LH^*_{RS}, nous avons proposé une structure de données adéquate faisant la correspondance entre les données source et les données de parité par une gestion optimisée des rangs des enregistrements. Nous avons également proposé et implanté des scénarios de création de fichier, manipulation client (insertion, suppression et mise à jour), récupération d'enregistrements et de cases et d'augmentation de la disponibilité du fichier par l'ajout de cases de parité. A cet effet, la

conception de chaque scénario est validée par des mesures de performances, et le scénario peut être transposé sur une autre architecture afin d'améliorer les performances.

Nous avons tenu à valider nos choix d'ordre architectural et d'ordre algorithmique, ainsi que les optimisations apportées tant au plan théorique qu'au plan expérimental. Nos réalisations pratiques se matérialisent par la mise au point d'un prototype, qui nous a permis de conduire des expérimentations, ayant pour but de mesurer les performances de nos réalisations. Les résultats sont prometteurs dans le domaine, et prouvent la performance de notre gestionnaire LH^*_{RS}. Notons que les scénarios proposés ne sont pas propres à LH^*_{RS}, et peuvent s'adapter à toute structure de données distribuée, indépendamment de l'algorithme de répartition des données et des algorithmes de codage/ décodage. Notre travail est plus ou moins inter-opérable avec tout algorithme de distribution de données et tout algorithme de codage/ décodage. Nos réalisations et conclusions concernent la conception de structures de données distribuées à haute disponibilité peuvent par ailleurs être exploitées.

Les résultats des travaux de recherche élaborés dans le cadre de cette thèse ont été publiés et démontrés au fur et à mesure de l'avancement dans la thèse [M02, ML02, LMS04, MS04, LMS04-*b*].

1.5 Plan de la Thèse

Cette thèse est organisée en deux parties. La première partie intitulée *Etat de l'Art*, situe cette thèse dans son contexte scientifique, et se consacre à l'état de l'art en matière de structures de données distribuées et à l'énumération de mécanismes de haute disponibilité. La deuxième partie s'intitule *LH^*_{RS} : Une SDDS à Haute Disponibilité* détaille les fondements théoriques, conception & la validation expérimentale de LH^*_{RS}.

Le *Chapitre II* décrit des notions fondamentales concernant les systèmes de stockage répartis : architecture matérielle, architecture de serveur, les SDDSs etc.

Le *chapitre III* énumère et détaille, à titre non exhaustif, des mécanismes de haute disponibilité proposés dans la littérature.

Le *chapitre IV*, présente les codes Reed Solomon, les fondements théoriques de la SDDS à haute disponibilité LH^*_{RS}. Nous présentons aussi notre nouveau code RS et détaillons les optimisations apportées au calcul de parité et à la récupération des données.

Vient ensuite, le *chapitre V* où nous proposons un gestionnaire LH^*_{RS} et nous décrivons les trois nouveaux composants (gestion des connexions TCP/IP, contrôle de flux & acquittement messages, et la table d'adresses dynamique) intégrés à l'architecture SDDS2000.

Dans le *chapitre VI*, nous proposons et évaluons la complexité des scénarios relatifs à la création du fichier LH^*_{RS}, l'augmentation de la haute disponibilité et la récupération de données.

Dans le *chapitre VII*, nous rapportons les mesures de performances des expérimentations menées afin d'évaluer nos choix architecturaux et conceptuels.

Enfin, le dernier chapitre intitulé *Conclusions & Travaux Futurs*, résume les apports de cette thèse et ouvre des voies de recherche.

PARTIE I : ETAT DE L'ART

CHAPITRE

2 INTRODUCTION AUX SYSTEMES REPARTIS

2.1 Introduction

L'évolution des architectures d'ordinateurs et des réseaux de communication, a conduit les chercheurs à repenser l'architecture des applications. Ces dernières doivent désormais tirer profit des nouvelles technologies. Dans la première section, nous décrivons les architectures parallèles. Dans la deuxième section, nous nous intéressons à l'architecture logicielle des systèmes répartis. La troisième section s'intéresse aux Structures de Données Distribuées et Scalables (SDDS). Enfin, nous présentons quelques éléments nécessaires à l'évaluation des performances d'un système réparti.

2.2 Architectures Parallèles

Dans ce qui suit, nous présentons trois architectures parallèles, définies selon le critère de partage de ressources et les architectures hybrides, telles qu'elles sont décrites dans la littérature [DG92, NZT96, ÖV99].

2.2.1 Architecture à Mémoires Partagées

Dans l'architecture à mémoires partagées (Figure 2-1-(*a*), ang. *Shared-Memory Architecture*), les disques et les mémoires centrales sont partagés par tous les processeurs du système. L'espace d'adressage global rend le système facile à implanter pour les vendeurs de SGBDs. La communication inter-processeurs est rapide, vu l'accès partagé aux mémoires centrales. De surcroît, l'équilibre de charge entre les processeurs est facile à réaliser, et l'échec d'un processeur n'entraîne pas la non-possibilité d'accès aux données. En contrepartie, le système est coûteux, et les accès conflictuels aux mémoires centrales peuvent dégrader les performances. Pour ce, le nombre de processeurs est limité à 20-30. En effet, un nombre supérieur crée des goulots d'étranglement.

Nous citons en exemples de systèmes de gestion de bases de données réparties sur SMP : Sequent, Encore, Escala, XPRS de l'université de Berkeley, DBS3 de Bull.

2.2.2 Architecture à Disques Partagés

Dans l'architecture à disques partagés (Figure 2-1-(*b*), ang. *Shared-Disk Architecture*), chaque processeur a sa mémoire centrale privée, mais les disques sont partagés par tous les processeurs du système. Cette architecture, en plus des avantages qu'elle hérite de l'architecture à mémoires partagées, réduit les accès conflictuels aux mémoires centrales. Par contre, son inconvénient majeur est lié à la complexité du maintien de la cohérence des caches des processeurs, par rapport aux données sauvées sur les disques.

Nous citons en exemples de systèmes de gestion de bases de données réparties : VAXcluster de DEC, IMS/VS Data Sharing de IBM, Oracle...

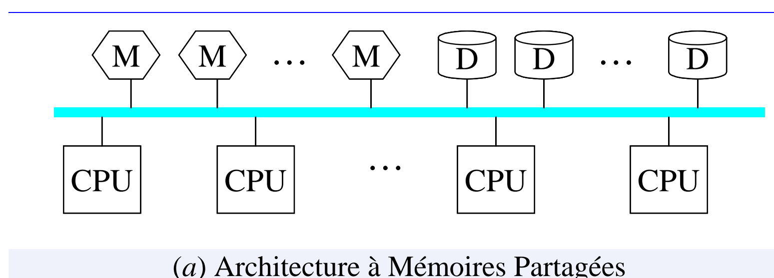
2.2.3 Architecture à Mémoires Distribuées

Dans l'architecture à mémoires distribuées (Figure 2-1-(*c*), ang. *Shared-Nothing Architecture*), chaque processeur a sa propre mémoire centrale et disque. Cette architecture est facilement extensible, et a un coût abordable vu que le système est une collection de postes de travail. Ces derniers communiquent par envoi de messages dans le réseau. Cette architecture est vulnérable aux problèmes d'échec de nœud et d'équilibre de charge. Contrairement aux architectures décrites précédemment, il y a nécessité de modifier les applications pour tirer profit de l'augmentation de puissance procurée.

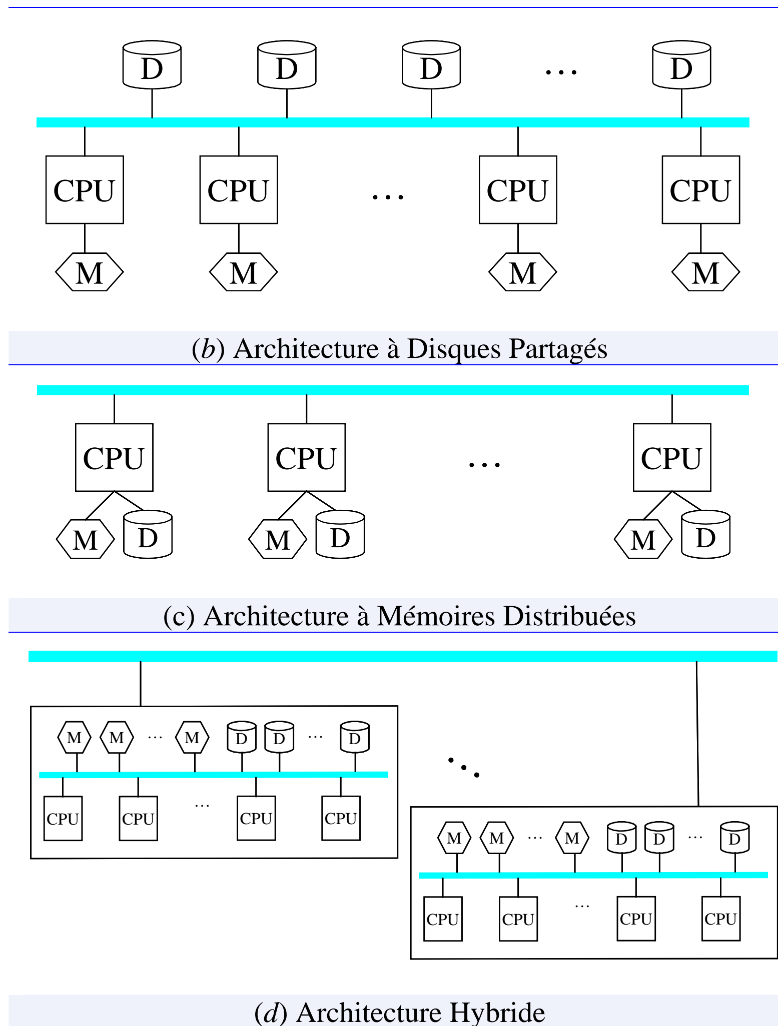
Nous citons en exemples de systèmes de gestion de bases de données réparties : Teradata de ATT, NonStopSQL de Tandem, Gamma de l'université de Wisconsin et Bubba de MCC.

2.2.4 Architectures hybrides

Une architecture hybride à deux niveaux, peut être au niveau interne une architecture à mémoire partagée, et au niveau externe une architecture à mémoires distribuées. De telles architectures combinent les avantages de chaque architecture, et compensent les inconvénients respectifs des architectures. L'architecture hybride illustrée dans la Figure 2-1-(*d*) combine l'équilibre de charge des architectures à mémoire partagée et l'extensibilité des architectures à mémoire distribuée.



(*a*) Architecture à Mémoires Partagées



(*b*) Architecture à Disques Partagés

(c) Architecture à Mémoires Distribuées

(*d*) Architecture Hybride

Figure 2-1: Architectures Matérielles Parallèles.

Une autre terminologie existe, elle classifie les architectures selon le degré de couplage. Ainsi, nous distinguons (*1*) les architectures à couplage serré correspondant aux SMPs : (acronyme de *Symmetric MultiProcessor Systems*) où un ensemble de processeurs partageant toutes les ressources du système; et (*2*) les architectures à couplage lâche correspondant aux MPPs (acronyme de *Massively Parallel Processor*) et aux Serveurs en grappes (ang. *cluster*). Un cluster est une interconnexion de systèmes indépendants ou nœuds par un réseau local, et tel que chaque nœud possède ses propres ressources et le mode de communication est l'échange de messages.

2.3 Distribution de Données & Traitement Parallèle

Dans cette section, nous évoquons en premier lieu des stratégies de distribution de données sur des architectures matérielles parallèles décrites dans la section précédente (§2.2). En deuxième lieu, nous décrivons des stratégies de traitement parallèle sur des données distribuées.

2.3.1 Stratégies de Distribution de Données

2.3.1.1 Distribution par 'Donneur de Cartes'

La stratégie de distribution selon le 'Donneur de Cartes' (ang. *Round-Robin Partitioning*) - illustrée dans la Figure 2-2-(*a*), est la plus simple. Si *N* est le nombre de nœuds, le $i^{éme}$ article (ou unité de fragmentation) est alloué au nœud de numéro : *i* mod *N*. Cette stratégie garantit un équilibre optimum de charge entre les nœuds et convient aux requêtes nécessitant un parcours séquentiel des articles, et permet de les paralléliser. Elle est implantée dans les systèmes RAID [PGK88].

2.3.1.2 Distribution par Hachage

La stratégie de distribution par hachage (ang. *Hashage Partitioning*) –illustrée dans la Figure 2-2-(*b*), distribue l'ensemble des articles en utilisant une fonction de hachage *h* sur un ensemble d'attributs. La fonction de hachage retourne le numéro de serveur, auquel l'article sera alloué. Elle convient aux requêtes, dont le prédicat implique les attributs de partitionnement.

2.3.1.3 Distribution par Intervalle

La stratégie de distribution par intervalles (ang. *Range Partitioning*) –illustrée dans la Figure 2-2-(*c*), distribue les articles en fonction de la valeur d'un ou plusieurs attributs. Elle convient aux requêtes dont le prédicat implique les attributs de partitionnement et donc les requêtes par intervalle. Mais, ne garantit pas un équilibre de charge entre les nœuds.

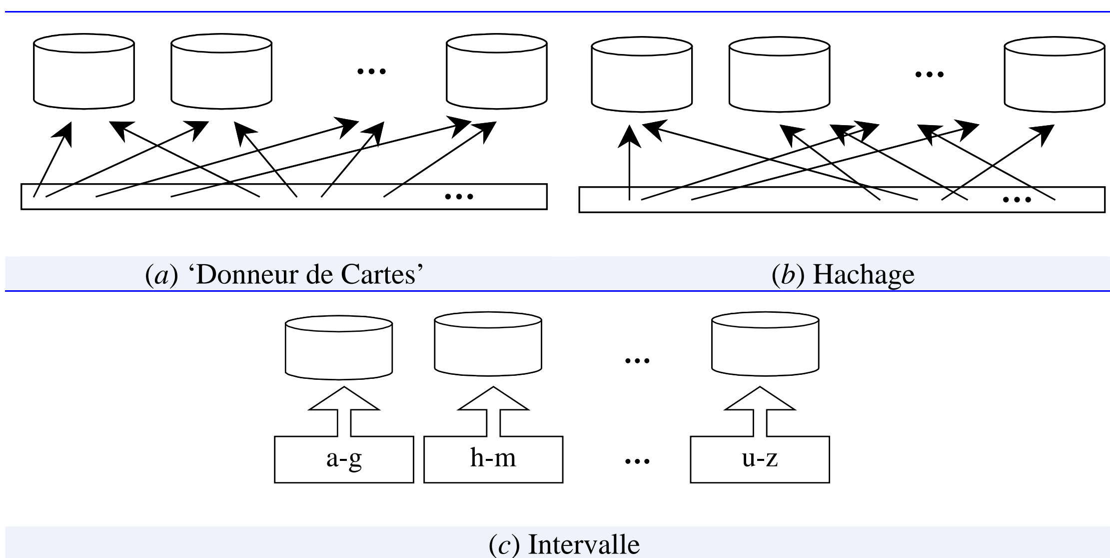


Figure 2-2 : Stratégies de Distribution des Données.

2.3.2 Stratégies de Traitement Parallèle

Dans le but d'avoir de meilleurs temps de réponse, des stratégies de traitement parallèles ont été développées. Nous distinguons deux types de parallélisme, que sont (*1*) le parallélisme inter-opérations et (*2*) le parallélisme intra-opérations. Dans le premier cas, plusieurs opérations s'exécutent en parallèle par différents processus. Alors que dans le deuxième cas, l'opération est sub-divisée en sous-opérations qui

s'exécutent en parallèle. Cela nécessite à la fin une fusion des résultats pour produire un résultat final

En ce qui concerne le parallélisme intra-opérations, des stratégies de traitement parallèle –illustrées dans la Figure 2-3, peuvent être appliquées pour réduire le temps d'exécution d'une opération, que sont :

- Parallélisme indépendant : (ang. *Independent Parallelism*) Il s'agit d'exécuter en parallèle plusieurs opérations indépendantes d'une même tâche ou requête.
- Parallélisme en tuyau : (ang. *Pipelined Parallelism*) C'est le cas où le résultat d'une opération constitue les données d'entrée pour l'opération suivante.
- Parallélisme par division : (ang. *Partitioned Parallelism*) Il s'agit de fragmenter la charge de travail entre plusieurs processus. Cela nécessite à la fin une fusion des résultats partiels pour produire un résultat final.

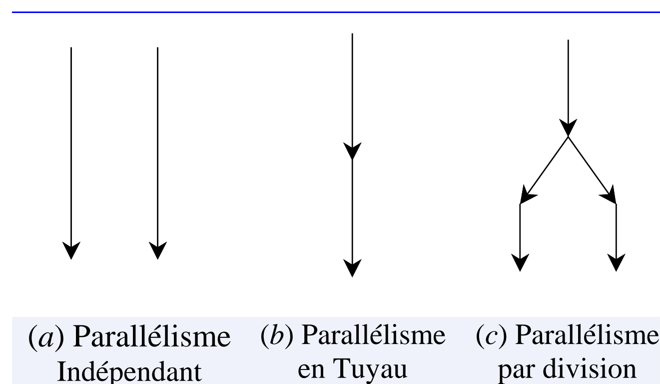


Figure 2-3 : Stratégies de Traitement Parallèle.

2.4 Architectures de Systèmes

Dans ce qui suit, nous décrivons les différentes architectures globales appliquées aux systèmes répartis par référence à [G94, GG96, P04], puis nous nous intéressons à l'architecture des serveurs.

2.4.1 Architecture Globale

2.4.1.1 Architecture Client/Serveur

C'est un modèle d'architecture où les programmes sont répartis entre processus clients et processus serveurs, communiquant par des requêtes-client en réponses-serveur. En général la machine *serveur* est puissante en terme de capacités d'entrée-sortie.

Un système client/serveur à 2 strates (ang. *two-tiered Client Server Architecture*) fonctionne comme illustré dans la Figure 2-4,

* Le *client* envoie une requête au *serveur* sachant son adresse IP et son port d'écoute spécifique au service demandé.

* Le *serveur* reçoit la demande, et répond, sachant l'adresse de la machine *client* et son port d'écoute.

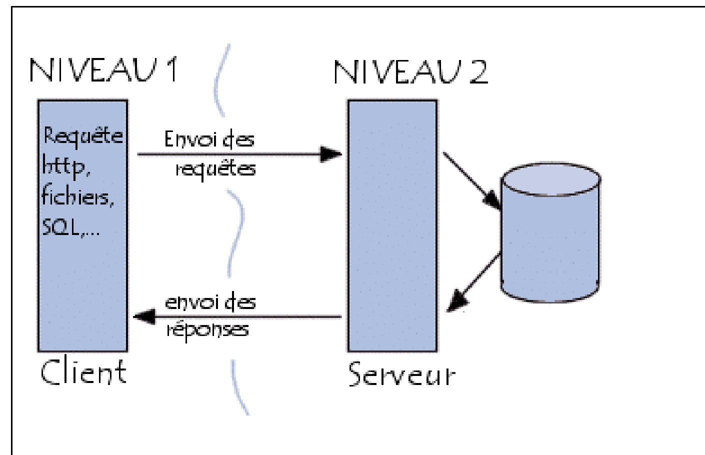


Figure 2-4 : Architecture Client Serveur à 2 Strates.

2.4.1.2 Architecture Pair à Pair

L'architecture *Pair à Pair* (ang. *Peer to Peer*, *P2P*) tire profit des ressources informatiques par agrégation des puissances de calcul et/ou des capacités de stockage disponibles dans un réseau. Par exemple, *Napster* et *Gnutella* agrégent les capacités de stockage, alors que *SETI@home* (acronyme de *Search for Extraterristrial Intelligence at Home*) [S01] agrège les puissances de calcul.

Bolosky et al. [BDET00] ont testé la faisabilité d'un système de gestion de fichiers distribué sur un ensemble de postes de travail de Microsoft Corp.. Ils ont mesuré l'espace disque non utilisé, la disponibilité des machines et la charge des machines du campus ; et ils ont conclu que (*1*) seule la moitié de l'espace disque est utilisée, (*2*) la moitié des machines est accessible 95% du temps, enfin (*3*) la plupart des machines sont non utilisées la majorité du temps. Ces résultats viennent à l'appui de la faisabilité d'installation d'un système de gestion de fichiers distribué déployé sur l'ensemble des ordinateurs d'une entreprise.

2.4.2 Architectures de Serveurs

Un intérêt croissant envers la programmation distribuée et l'implantation de serveurs scalables et performants a poussé les chercheurs à classifier les différentes architectures serveurs. En effet, au moment de la conception d'un serveur, il y a le choix d'implanter le serveur en tant que *mono-thread* (ang. *single threaded*) ou *multi-thread* (ang. *multithreaded*). Les enjeux sont exprimés en termes de besoins ou charge de travail, performance ou rapidité de traitement, difficulté de programmation et de débogage et concurrence entre les *threads*. La décision est conséquente, et ce n'est que par l'expérimentation que l'architecture *thread* optimale est déterminée.

Lorsque la charge de travail d'un serveur croit, une architecture non modulaire dégrade les performances. Pour ce, des stratégies de *threading* dynamiques ont été définies afin d'avoir une architecture serveur scalable. Elles consistent à réguler le nombre de *threads* et leurs modes opératoires en fonction de la charge de travail à laquelle fait face le serveur.

2.4.2.1 Création & Emploi des *Threads*

Deux stratégies existent pour la création de *threads* ou processus (ang. *Thread/ Process Spawning*) que sont:

Stratégie de création à la demande (ang. *Request Spawning*): Pour chaque nouvelle requête un *thread/ processus* est créé. Cette stratégie a pour conséquence des coûts de création et de terminaison de *thread* influant le temps de réponse.

Stratégie de pré-création (ang. *Pre-Spawning* ou *Thread/ Process Pool*): Un ensemble de *threads/ processus* prêts à l'emploi est crée pour traiter les requêtes, et ce pour amortir le coût de création et de destruction de *threads /procesus*.

Deux stratégies existent pour allouer les requêtes du client aux *threads / Processus* que sont : (*1*) Allouer un *thread/ processus* par requête (ang. *Thread/ Process-per-Request strategy*) ou (*2*) Allouer un *thread/ processus* par session (ang. *Thread/ Process-per-Session strategy*). D'après Hu, Pyarali et Schmidt [HPS00], le temps de création d'un processus est d'un ordre de magnitude supérieur au temps de création d'un *thread*.

Stratégie Thread-par-Requête: Chaque requête-client est prise en charge par un *thread* désigné pour traiter la requête client, et ce indépendamment du fait que d'autres requêtes du même client pourraient parvenir et soient les suivantes requêtes à traiter.

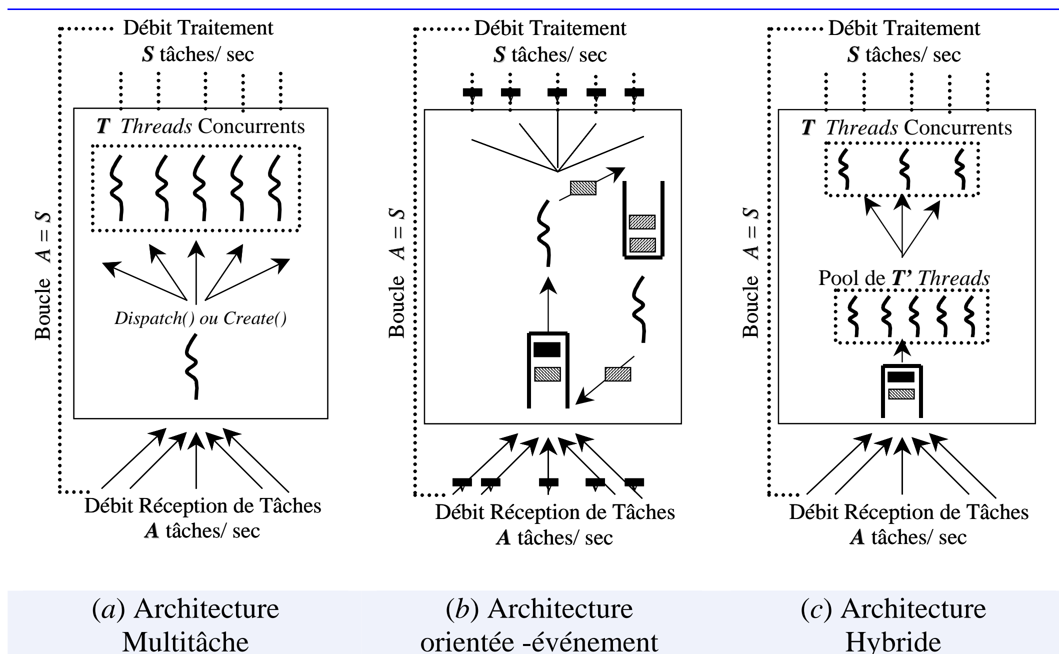
Stratégie Thread-par-Session: Cette stratégie s'applique au cas où de multiples requêtes parviennent du même client, ainsi une session client est ouverte. Tel que, tant que la session est ouverte le même *thread* servirait pour traiter toutes les requêtes du client initiées dans la session.

2.4.2.2 Classification d'Architectures Serveurs

Dans [WGBC00], Welch et al. évoquent trois types d'architectures, que sont (*1*) les Architectures Multitâches (ang. *Multithreaded Architecture*), (*2*) les Architectures orientées-événement (ang. *Event-driven Architectures*) et (*3*) les Architectures Hybrides (ang. *Hybrid Architectures*).

Architecture Multitâche: Elle est illustrée dans la Figure 2-5-(*a*). Il s'agit de *threads* créés soit en pool ou à la demande. La commutation entre les *threads* est gérée par le système d'exploitation donc transparente au programmeur. Notons que, plusieurs bibliothèques de langages de programmation modernes offrent des APIs pour la manipulation de *threads* ainsi que des objets de synchronisation entre des *threads* concurrents. D'après Welch et al. [WGBC00], indépendamment de la conception du serveur multitâche, quand le nombre de *threads* augmente, le monitorat des *threads* devient pénalisant pour le système d'exploitation. Par conséquent, il existe un seuil maximum *T'* du nombre de *threads*, au-delà duquel les performances se dégradent. La Figure 2-6-(*a*) montre la variation du débit serveur en fonction du nombre de *threads*.

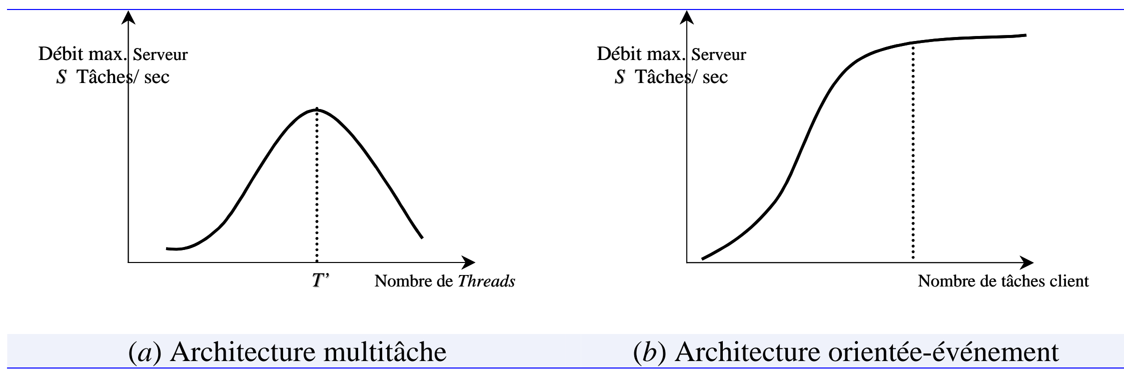
Architecture orientée-événement: Elle est illustrée dans la Figure 2-5-(*b*). Il s'agit d'un *thread* qui empile des 'évènements' à traiter de la pile communication, et d'un second *thread* qui dépile de la file les événements afin de les traiter. Cette architecture fournit un haut débit de traitement qui dépend du nombre de tâches à exécuter. Vu l'absence de concurrence entre les *threads*, l'architecture ne traite pas de verrous mortels et de situations de concurrence et est robuste face à une montée de la charge car l'excès de charge est absorbé par les piles communication et la file des 'évènements'. Le mode de programmation de ce type d'architectures est peu développé, et ne profite pas des configurations multiprocesseurs. Remarquons que dès la saturation du système, la courbe de performances s'aplatit. Il n'empêche que les performances ne se dégradent pas. D'après les expérimentations menées par Welch et al. [WGBC00], ce type d'architecture offre de meilleures performances que les architectures multitâches, comme le montre la Figure 2-6-(*b*).



(*a*) Architecture Multitâche | (*b*) Architecture orientée -événement | (*c*) Architecture Hybride

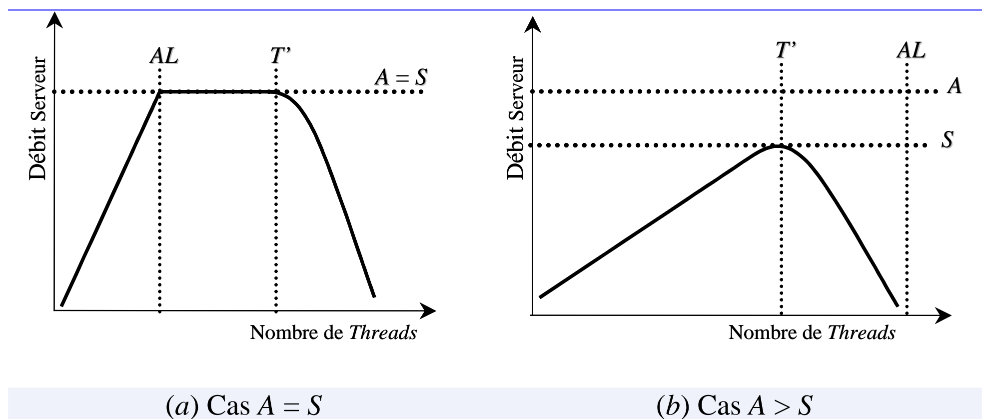
Figure 2-5 : Classification des Architectures selon Welch et al. [WGBC00].

Architectures hybrides: Elle est illustrée dans la Figure 2-5-(*c*). Ce type d'architectures est dans le spectre ayant pour extrémités d'une part les architectures multitâches et d'autre part les architectures orientées-événements. Elles combinent ainsi le principe multitâche et les files pour absorber l'extra flux. Théoriquement, nous distinguons trois cas déterminant l'allure de la courbe de performances d'une architecture hybride, étant le cas où A est égal à S, le cas où A est strictement supérieur à S, et le cas où A est strictement inférieur à S. Dans le dernier cas, le système ne connaît pas de problèmes de performance.



(*a*) Architecture multitâche (*b*) Architecture orientée-événement

Figure 2-6 : Allure des courbes de performances des architectures multi-tâches et des architectures orientées-événement.



(*a*) Cas $A = S$ (*b*) Cas $A > S$

Figure 2-7 : Allure des courbes de performances d'une architecture hybride.

2.4.2.3 Les Limites du Parallélisme

Plusieurs facteurs peuvent influer sur le temps d'exécution d'une tâche exécutée par plusieurs processus, que sont :

Démarrage (ang. *Start up*): C'est le temps nécessaire pour démarrer une opération parallèle, qui peut dominer le temps de calcul.

Interférence (ang. *Interference*): Lors de l'accès à une ressource partagée, un processus ralentit les autres.

Biais (ang. *Skew*): Le temps de réponse de plusieurs processus parallèles est le temps de réponse du processus le plus lent.

2.5 La Communication Réseau

Plusieurs langages de programmation modernes offrent des APIs pour la communication réseau, que ce soit dans l'environnement Unix par les *BSD Sockets* ou dans l'environnement Windows par les *WinSockets*.

La notion de *sockets* a été introduite dans les distributions de Berkeley. C'est la raison pour laquelle on parle de *sockets* BSD (*Berkeley Software Distribution*). Il s'agit d'un modèle permettant la communication inter-processus (ang. *IPC- Inter-Process Communication*) afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un réseau.

Comme le montre la Figure 2-8, les *sockets* se situent juste au-dessus de la couche de Transport du modèle OSI.

Modèle de sockets	***Modèle OSI***
Application utilisant les *sockets*	Application
	Présentation
	Session
UDP, TCP, …	Transport
IP, ARP	Réseau
Ethernet, X25, ...	Liaison
	Physique

Figure 2-8 : Position des *Sockets* dans le modèle OSI.

Dans ce qui suit, nous nous intéressons à la famille IP (ang. *Internet Protocol*) et plus particulièrement aux protocoles UDP et TCP/IP.

2.5.1 Protocole UDP

Le protocole UDP -acronyme de *User Datagram Protocol*, est un protocole non fiable qui opère en mode non connecté. Le protocole porte le qualificatif non fiable, car l'application l'utilisant risque :

* La perte de paquets, ceci est du au fait que le protocole ne traite pas les accusés de réception.
* Le désordre, en effet l'ordre d'arrivée des paquets peut être différent de leurs ordres d'émission.
* La duplication de messages.

Même si UDP n'est pas fiable, il est plutôt rapide et pratique pour les applications où on préfère la vitesse à la fiabilité.

2.5.2 Protocole TCP

Contrairement au protocole UDP, le protocole TCP –acronyme de *Transfer Control Protocol*, opère en mode connecté et traite les cas de données perdues, dupliquées, ou arrivées dans le désordre à l'autre bout de la liaison Internet. Le revers de la médaille étant une lenteur par rapport à UDP.

Le mécanisme de fiabilité de TCP est assuré par l'insertion d'un numéro de séquence, et par l'obligation d'émission d'un accusé de réception (ACK) par le TCP destinataire. Si l'accusé de réception n'est pas reçu au bout d'un temps prédéfini, le paquet sera ré-émis. Côté récepteur, les numéros de séquence sont utilisés pour reconstituer dans le bon ordre le flux original et éliminer les paquets dupliqués. Une connexion TCP est bidirectionnelle et prend la dénomination de *full-duplex*.

TCP fournit un moyen au destinataire pour contrôler le débit de données envoyé par l'émetteur. Ceci est obtenu en retournant une information de *fenêtre* avec chaque accusé de réception indiquant la capacité de réception instantanée en termes de numéros de séquence. Le paramètre *fenêtre* indique le nombre d'octets que l'émetteur peut envoyer avant une autorisation d'émettre ultérieure.

2.6 Les Structures de Données Distribuées et Scalables

Une Structure de Données Distribuée et Scalable satisfait les trois propriétés clé suivantes:

1. La première propriété consiste en *l'absence d'un répertoire d'accès centralisé*. Son existence aurait créé un goulot d'étranglement, et par conséquent une dégradation des performances d'accès.
2. La deuxième propriété fondamentale de toute SDDS est que le fichier correspondant *s'étend* sur plusieurs sites et *se rétrécit de manière dynamique et transparente pour l'application*, et ce respectivement, à travers des éclatements et des fusions de cases.
3. Enfin, *chaque site a une image du fichier* et est capable de *détecter une erreur* d'adressage, et de *rediriger la requête* vers un autre site.

Plusieurs SDDSs ont été proposées, différant par leurs stratégies de distribution de données. La première famille est basée sur la distribution par hachage: LH* évoquée dans la section §2.3.1. La deuxième RP* implante la distribution par intervalles : *Range Partitioning* évoquée dans la section §2.3.1.

2.6.1 Distribution par Hachage

Plusieurs structures de données distribuées ont utilisé le hachage comme méthode d'accès à des données distribuées, notamment LH* [LNS93], DDH [D93], EH* [HBC97]. La SDDS, LH* a fait l'objet de plusieurs extensions et implantations, nous citons à titre non exhaustif les travaux de Devine [D93], Vingralek et al. [VBW94, VBW95], Karlesson et al. [KLR96], Hilford et al. [HBC97] et Bennour [B00]. Dans ce qui suit, nous décrivons ces différents schémas.

2.6.1.1 LH*

La structure de données distribuée et scalable LH* est due à Litwin, Neimat et Schneider [LNS93, LNS96]. Cette SDDS utilise le hachage linéaire [L80] comme stratégie de distribution des enregistrements de données sur un ensemble de machines. Dans ce qui suit, nous rappelons brièvement le principe du hachage linéaire.

*** Hachage Linéaire**

Un fichier LH est une collection de cases, contenant des enregistrements. Le fichier s'étend et se rétrécit de manière dynamique, et ce respectivement, à travers des éclatements et des fusions de cases.

L'Algorithme 2-1 montre qu'un enregistrement est alloué à une case, dont l'adresse logique est déterminée par le calcul de la valeur de la fonction de hachage $h_i(c) = c$ modulo $N^* 2^i$, tels que c désigne la clé de l'enregistrement, N le nombre de cases initiales, i le niveau du fichier et n dénote un pointeur sur la prochaine case à éclater.

Algorithme 2-1 : Adressage logique LH d'un enregistrement de clé c.

Entrées: (i, n), c
$a \leftarrow h_i(c)$
Si $(a < n)$ alors $a \leftarrow h_{i+1}(c)$ Fin Si

Les cases ont une capacité de *b* enregistrements, quand le contenu d'une case excède *b*, une case *n* éclate. Les éclatements se font dans un ordre déterministe : 0 .. *N*-1, 0 .. 2*N-1, … etc. Suite à un éclatement de la case *n*, la moitié des enregistrements migrent vers une nouvelle case créée de numéro logique $n + N*2^i$, et le couple (*i*, *n*) caractérisant le fichier LH change selon Algorithme 2-2.

Algorithme 2-2 : Mise à jour du couple (*i*, *n*) après un éclatement d'une case.

Entrées: (*i'*, *n'*)
$n \leftarrow n + 1$
Si $n \geq 2^i$ alors $n \leftarrow 0$ et $i \leftarrow i + 1$ Fin Si

Sans contrôle d'éclatements, le taux de remplissage du fichier est de 65-69% [LNS93, B00], il peut être amélioré, en ajoutant des contraintes conditionnant l'éclatement des cases, par exemple la procédure d'éclatement n'est faite que si le taux de remplissage actuel est au moins égal à un certain seuil *t*.

Des suppressions dans le fichier, diminuent le taux de remplissage, et déclenchent une fusion de cases, qui est le phénomène inverse de l'éclatement. Dans ce cas, le couple (*i, n*) évolue selon Algorithme 2-3.

Algorithme 2-3 : Mise à jour du couple (*i*, *n*) après la fusion de deux cases.

Entrées: (*i, n*).
$n \leftarrow n - 1$
Si $(n < 0)$ alors $i \leftarrow i$ -1 et $n \leftarrow 2^i - 1$ Fin Si

*** Architecture**

Cette SDDS, comme le montre la Figure 2-9, implique trois acteurs que sont : cases, clients et le coordinateur. Certaines variantes ne supportent pas le coordinateur. Dans ce qui suit, nous détaillons le schéma LH* au niveau de chacun du client, une case LH* et le coordinateur.

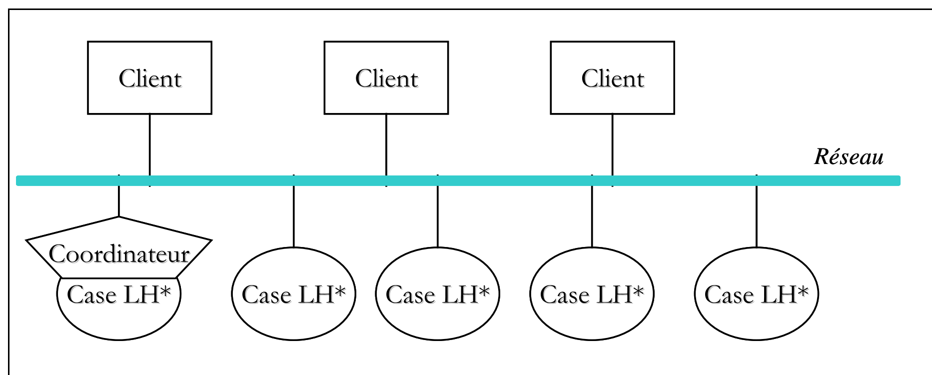


Figure 2-9 : Illustration de la SDDS LH*.

*** Client**

L'algorithme de hachage linéaire (§2.6.1.1), suppose qu'un client utilise le couple (*i, n*) pour le calcul de l'adresse logique d'un enregistrement de clé *c*. Vu qu'il s'agit d'une donnée partagée, il serait requis de mettre à jour le couple (*i, n*), après chaque éclatement de case, et pour chaque client. Pour éviter cela, le client a sa propre image

du client du fichier (i', n'), et suivant laquelle il calcule l'adresse a' de l'enregistrement c (voir Algorithme 2-4).

Algorithme 2-4 : Adressage exécuté par le client.

Entrées: (i', n'), c.
$a' \leftarrow h_{i'}(c)$
Si ($a' < n'$) alors $a' \leftarrow h_{i'+1}(c)$ Fin Si

L'image du client n'est pas nécessairement correcte. La case LH* de numéro logique a' ou une autre case LH* renverrait au client un message d'ajustement d'image. Le message permet au client d'actualiser son image, il comporte j et a, étant le niveau de la case et l'adresse logique de la case correcte. Notons que, après exécution de l'Algorithme 2-5, l'image du client ne devient pas forcément identique à (i, n).

Algorithme 2-5 : Ajustement de l'image client.

Entrées: (i', n'), j, a.
$i' \leftarrow j - 1$
$n' \leftarrow a + 1$
Si $n' \geq 2^{i'}$ alors $n' \leftarrow 0$ et $i' \leftarrow i' + 1$ Fin Si

*** Case LH***

A la réception d'une requête, une case LH* commence par vérifier si la requête lui est destinée ou non, et donc traite la requête ou la renvoie à une autre case LH*. Le schéma LH* garantit qu'une requête atteint le bon serveur au maximum au bout de deux renvois (voir [LNS93] pour les démonstrations).

Algorithme 2-6 : Test & Renvoi exécuté par la case.

Entrées: c, j, a.
$a' \leftarrow h_j(c)$
Si ($a' \neq a$) Alors
    $a'' \leftarrow h_{j-1}(c)$
    Si ($a'' \in]a, a'[$) Alors $a' \leftarrow a''$ Fin Si
    Renvoyer c à a'
Sinon Traiter Requête(c)
Fin Si

*** Coordinateur**

Le coordinateur coordonne les éclatements des cases LH*, et donc est le seul à avoir une image correcte du fichier. Il est implanté au-dessus de la première case LH*. Quand une case LH* déborde, son contenu en nombre d'enregistrements dépasse b, elle le notifie au coordinateur. Les éclatements se font dans un ordre déterministe : 0 .. N-1, 0 .. 2*N-1, … etc.. Le coordinateur initie la procédure d'éclatement de la case n. L'éclatement consiste en le transfert d'approximativement la moitié du contenu de la case n vers une nouvelle case.

2.6.1.2 DDH

Devine [D93] soulève quatre inconvénients du schéma de distribution de données LH* et propose la SDDS : DDH (*Dynamic Distributed Hashing*) se basant sur

l'algorithme de hachage dynamique proposé par Larson [L78]. Les quatre inconvénients soulevés sont (*1*) LH* a été proposée tel que un nœud héberge une seule case, ceci oblige à concevoir des cases volumineuses, ce qui dégrade les performances. Devine propose que plusieurs cases puissent cohabiter sur le même nœud, en optant pour de petites cases gérées par un *serveur DDH*; (*2*) Le deuxième inconvénient pointe la perte d'un message important, tel que l'ordre d'éclatement. Ainsi, la nécessité de coordination entre les cases et le coordinateur qui se traduit par une relation de dépendance réduisant l'autonomie locale des serveurs; (*3*) Dans LH* un message arrive à sa destination au maximum au bout de deux renvois, et ce si l'image du client est fausse. D'après Devine ceci n'est pas vrai si la vitesse du client est inférieure à la vitesse des éclatements ; (*4*) Enfin, le schéma d'éclatement LH* est jugé inéquitable vu que les éclatements se font dans un ordre déterministe, et ce ne sont pas les cases surchargées qui éclatent. Ainsi, les cases surchargées doivent supporter un excès de charge pendant que des cases non surchargées éclatent. Le problème est que attendre un ordre d'éclatement nécessite la coordination entre les entités impliquées et une dépendance vis à vis de l'entité Coordinateur. Alors que l'éclatement autonome de cases crée des problèmes d'adressage.

Devine propose l'implantation sur chaque nœud d'un *Serveur DDH* qui permet au nœud d'être autonome. Cette autonomie permet au serveur de décider de l'éclatement/ fusion de cases, et lui évite l'échange de messages de coordination. Ainsi, le serveur DDH sauve les numéros des cases qui sont sous sa tutelle. Il réceptionne aussi les messages à l'intention des cases, contrôle la charge du nœud, et maintient un répertoire d'adresses numéro case LH* - serveur DDH.

Devine propose aussi une méthode de calcul des adresses beaucoup moins coûteuse que le *modulo* en utilisant le ET binaire (&[1]: ang. *bitwise AND*) et l'opérateur de décalage de bits à gauche (<<[2] : ang. *bitwise left shift*). Ainsi, $h_i(c) = c \bmod 2^i$ est remplacée par $h_i(c) = c \,\&\, ((1 << i) - 1)$.

Le prototype utilise le protocole UDP pour l'envoi/ réception de messages. En cas de perte d'un message détectée par la non-arrivée de la réponse, le message est retransmis. Une étude expérimentale a été menée pour valider le schéma proposé.

2.6.1.3 Travaux de Vingralek et al.

Vingralek, Breitbart et Weikum [VBW94, VBW98] proposent un schéma de distribution de données, où plusieurs cases sont allouées à un nœud. Ainsi, plusieurs cases LH* peuvent cohabiter sur le même nœud, et ce dans le but de minimiser le nombre de nœuds sur lesquels le fichier s'étend, par souci d'administration et de disponibilité de nœuds. Contrairement à LH* [LNS93] et DDH [D93], le schéma de distribution de données proposé tient compte du taux de remplissage des cases. La redistribution de la charge s'effectue alors de deux manières soit par l'éclatement de cases ou migration de cases. Le choix entre les deux alternatives est décidé par le coordinateur. Ce dernier exécute une heuristique, qui lui fournit une idée approximative sur l'occupation des cases. Notons que la première stratégie de

---

[1] Le ET binaire (&) compare chaque bit du premier opérande au bit de même rang du deuxième opérande, si les deux bits sont égaux à 1, le bit résultat est 1 sinon 0.

[2] L'opérateur de décalage de bits décale le premier opérande à gauche (<<) ou à droite (>>) par le nombre de positions indiqué dans le deuxième opérande.

redistribution de charge consiste en la migration de cases du serveur le plus surchargé vers le serveur le moins chargé ; et que la deuxième stratégie crée un nouveau serveur et transfère la moitié des enregistrements du serveur surchargé vers le nouveau serveur. Le choix entre les deux alternatives est décidé par l'entité *file advisor* – l'équivalent du *coordinateur* dans LH*. Ce dernier, à la réception d'une notification de surcharge d'un serveur, estime le taux de remplissage du fichier. Si le taux de remplissage est inférieur à un seuil *U*, il y a juste migration de cases d'un serveur à un autre, sinon éclatement de serveur.

Au niveau de chaque nœud est implanté un contrôle local de charge, qui prévient contre la surcharge du serveur. Ainsi, quand un serveur atteint la capacité maximale, il avertit le *file advisor*, et le relance jusqu'à réception du message d'éclatement ou de migration. Les relances se font après insertion de *x* enregistrements. Cette stratégie de relance est très intéressante, pour le cadre de LH*. En effet, sans ce, les cases LH* surchargées et ne recevant pas d'éclatement, avertiront le coordinateur de leur surcharge à chaque insertion d'un enregistrement. Ceci sature le réseau par des messages, surcharge le coordinateur par des messages à traiter, et risque même de déréguler l'ordonnancement des éclatements. Vingralek et al. ont mené une étude expérimentale pour valider le schéma proposé.

2.6.1.4 LH*$_{LH}$

Les travaux de Karlesson, Litwin et Risch [KLR96] spécifient l'organisation interne d'une case LH*. Ainsi, ils proposent une organisation LH pour une case LH*. Le schéma LH*$_{LH}$ utilise alors deux niveaux d'indexation. Le premier est un niveau d'indexation réseau géré par l'algorithme LH* qui permet au client de trouver la case LH* pour exécuter une requête. Le deuxième niveau est un niveau d'indexation interne géré par l'algorithme LH. La structure d'une case LH*$_{LH}$ est illustrée dans la Figure 2-10.

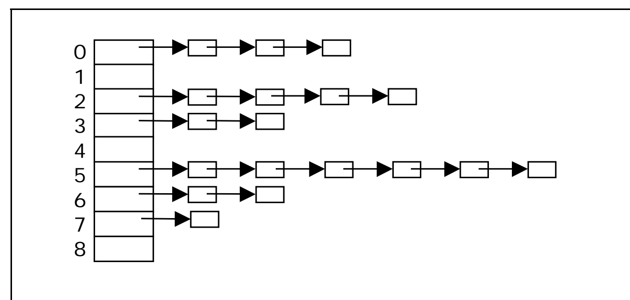


Figure 2-10 : Structure d'une Case LH*$_{LH}$.

2.6.1.5 Gestionnaire LH*$_{LH}$

Bennour a proposé et implanté un gestionnaire LH* pour LH*$_{LH}$ [BDNL00, B00, B02]. Une architecture de serveurs a été proposée et est basée sur le multitâche. Les messages sont réceptionnés et empilés dans une file par un *Thread d'écoute UDP*. Ils sont ensuite traités par un pool de *threads*. Les données sont sauvées au niveau de la mémoire centrale de chaque nœud en utilisant les fichiers mappés en mémoire (ang. *Memory Mapped Files*) [S96].

Par rapport à l'implantation de Devine [D93], Bennour a utilisé le protocole UDP pour la réception/envoi de messages dont la taille est inférieure à 400 octets, et le protocole TCP/IP par souci de performance et de fiabilité au moment de l'éclatement,

pour le transfert de la moitié du contenu la case qui éclate vers une nouvelle case. Une étude expérimentale a été menée pour valider les choix architecturaux.

2.6.1.6 Hachage Extensible Distribué

Hilford, Bastani et Cukic ont proposé une SDDS: EH* [HBC97]. EH* distribue les tuples en utilisant l'algorithme d'hachage extensible (ang. *Extendible Haching -EH*) proposé par Fagin et al. [FNPS79].

2.6.2 Distribution par Intervalle

En alternative à la distribution par hachage, les schémas de distribution par intervalle favorisent les requêtes par intervalles. Le schéma de distribution RP* (acronyme de *Range Partitioning*) fut proposé par Litwin, Neimat et Schneider [LNS94]. Plusieurs variantes de distribution par intervalle ont été investiguées, nous citons à titre non exhaustif, et sans les décrire : DRT* (acronyme de *Distributed Random Tree*) [KW94], BDST (acronyme de *Balanced and Distributed Search Trees*) [PN99], DSL [BM00], ADST [PN01], LDT (ang. *Logarithmic Distributed Serach Tree*) [BM02].

Dans ce qui suit, nous décrivons le schéma RP* [LNS94] et une proposition d'un gestionnaire RP* par Diéne [DL00, DL01, D01,D01-p].

2.6.2.1 RP*

Un fichier RP* [LNS94] est une collection de cases, contenant des enregistrements identifiés par un champ clef. Chaque case est organisée en arbre-B, et lui est associé un intervalle $]\lambda, \Lambda]$ déterminant l'ensemble des clefs d'enregistrements appartenant à la case. Ainsi, un enregistrement de clef c appartient à la case d -d'intervalle $]\lambda, \Lambda]$, si et seulement si $c \in\ ]\lambda, \Lambda]$.

Un fichier RP* est initialement crée sur une seule case, dite la case 0. cette dernière est d'intervalle $]-\infty, +\infty[$. Le fichier évolue à travers les éclatements de cases. En effet, chaque case RP* a une capacité maximum b d'enregistrements. Une case surchargée éclate. L'éclatement consiste à transférer approximativement la moitié du contenu de la case surchargée vers une nouvelle case RP*. La définition de la clé de division est stratégique pour l'équilibre de charge.

Plusieurs variantes du schéma RP* furent proposées, notamment RP^*_N, RP^*_C et RP^*_S. Ces variantes diffèrent par le protocole de communication utilisé pour s'adresser aux cases, étant soit multipoints (ang. *multicast*) ou point à point (ang. *unicast*). Le schéma RP^*_N utilise la communication multipoints pour toute manipulation du fichier. Ainsi, une requête d'insertion d'un enregistrement est envoyée à toutes les cases RP*, et seule une case la traite. Pour réduire le nombre de messages dans le réseau et la charge des cases, la variante RP^*_C implante une image du fichier au niveau des clients. Ces derniers désormais maintiennent une structure d'adressage leur permettant d'adresser directement leurs requêtes aux cases concernées, et à défaut utiliser la communication multipoint. Enfin, la variante RP^*_S n'utilise que la communication point à point pour les requêtes d'insertion, suppression, mise à jour et recherche. Pour ce, elle implante un index au niveau des cases.

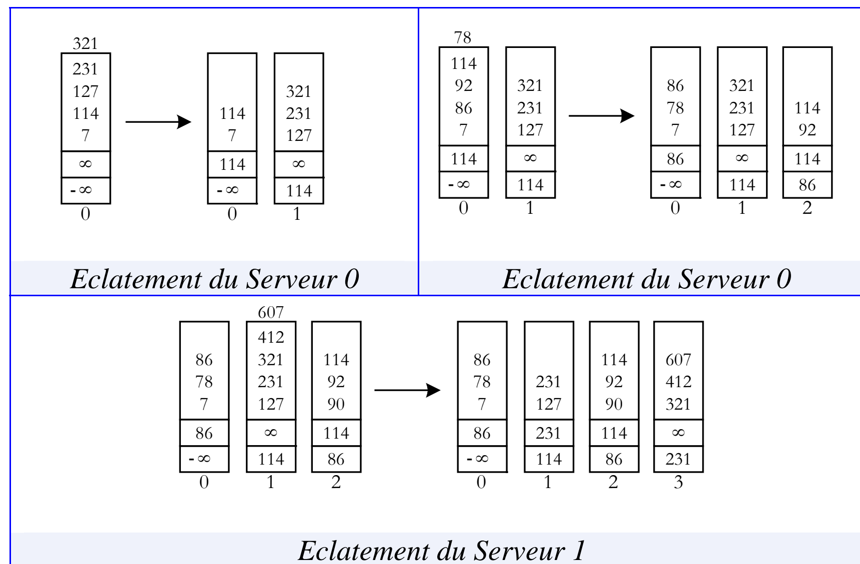


Figure 2-11 : Exemple d'évolution d'un fichier RP*.

2.6.2.2 Un Gestionnaire RP*

A. W. Diène [DL00, DL01, D01] a conçu et implanté un gestionnaire de fichiers RP*. Il a proposé également une architecture serveur permettant l'accès concurrent, la manipulation des articles du fichier RP* avec contrôle de flux et acquittement de messages. Une étude expérimentale approfondie a été faite pour comparer les performances des différentes variantes RP* implantées. Le prototype est téléchargeable du site web du CERIA [D01-p]. Les travaux de A. W. Diéne font l'objet de plusieurs travaux de recherche par extension.

2.7 Critères d'évaluation d'un Système Réparti

L'évaluation de la complexité d'un système réparti tient compte de la composante traitement local : le temps de traitement dépendant de la vitesse du processeur et des performances des autres composants matériels, tels que la mémoire centrale, le nombre d'E/Ss…. Outre cette composante, existe la composante de communication. Cette dernière dépend de la bande passante du réseau, elle est évaluée par le protocole de communication utilisé et le trafic sur le réseau -nombre et taille des messages ou tampons envoyés et reçus.

DeWitt et Gray [DG92], décrivent deux états de système limites qui sont :

* Saturation des Entrées/ Sorties (ang. *I/O bound*) ; cette situation est appelée *I/O bound*, parce que les performances du système sont limitées par les Entrées/ Sorties.
* Saturation des ressources de traitement (ang. *CPU bound*); cette situation est appelée *CPU bound*, parce que les performances du système sont limitées par la vitesse des processeurs.

Dans ce qui suit, nous commençons par énumérer les 12 critères établis par Date pour concevoir une application distribuée. Les sections suivantes décrivent deux métriques permettant d'évaluer des performances d'une application répartie par rapport à une

augmentation de la charge ou/et évolution de la configuration matérielle. Enfin, nous discutons les limites de nature intrinsèque–caractéristiques du milieu.

2.7.1 Les 12 Critères de Conception d'un Système Réparti de Date

C.J. Date a énuméré douze critères [D87] à satisfaire pour la conception et l'implantation de systèmes répartis, qui sont :

(1) *Autonomie de chaque site:* chaque site est responsable de l'intégrité de ses données, de sa propre sécurité et de sa propre gestion. Chaque site est fonctionnel même s'il ne peut pas communiquer avec les autres sites.

(2) *Absence de site privilégié:* l'objectif est de ne pas reposer sur un site unique pour éviter la centralisation de l'information.

(3) *Continuité de service:* le système réparti doit être tolérant aux pannes, tel que les pannes sont indépendantes et sans incidence.

(4) *Transparence vis à vis de la localisation des données:* à ce niveau de transparence les utilisateurs ne connaissent pas le schéma d'allocation des données.

(5) *Transparence vis à vis de la fragmentation:* par ce niveau de transparence les utilisateurs ne connaissent pas le schéma de fragmentation des données.

(6) *Transparence vis à vis de la réplication:* par ce niveau de transparence les utilisateurs ne connaissent pas si les données sont répliquées ou non.

(7) *Traitement des requêtes distribuées:* le support de requêtes distribuées est essentiel, ainsi que des méthodes d'optimisation des exécutions de requêtes en milieu réparti.

(8) *Traitement des transactions distribuées:* le support de transactions distribuées est essentiel, ainsi que des méthodes d'optimisation des exécutions de transactions en milieu réparti.

(9) *Indépendance vis à vis du matériel:* s'exécute sur différentes architectures matérielles.

(10) *Indépendance vis à vis du système d'exploitation:* le système peut être utilisé dans différents environnements.

(11) *Indépendance vis à vis du réseau:* utiliser un protocole réseau permettant de s'affranchir des différents types de réseaux.

(12) *Indépendance vis à vis du logiciel:* un système inter-opérable du point de vue logiciel possède des interfaces de communication avec d'autres logiciels.

2.7.2 Facteur de Rapidité

Le facteur de rapidité ou facteur d'accélération (ang. *speed-up*), est le temps mis par un processeur pour exécuter une requête divisé par le temps mis par *n* processeurs pour exécuter la même requête. Ce facteur mesure la diminution du temps de réponse en augmentant le nombre de nœuds pour l'exécution d'une même requête sur un même volume de données.

$$Facteur\ de\ Rapidité = \frac{Temps\ mis\ par\ le\ petit\ système}{Temps\ mis\ par\ le\ gros\ système}$$

La Figure 2-12 montre qu'une requête exécutée en quatre minutes par un processeur, et en seulement une minute par quatre processeurs. Le facteur de rapidité est alors égal à 4.

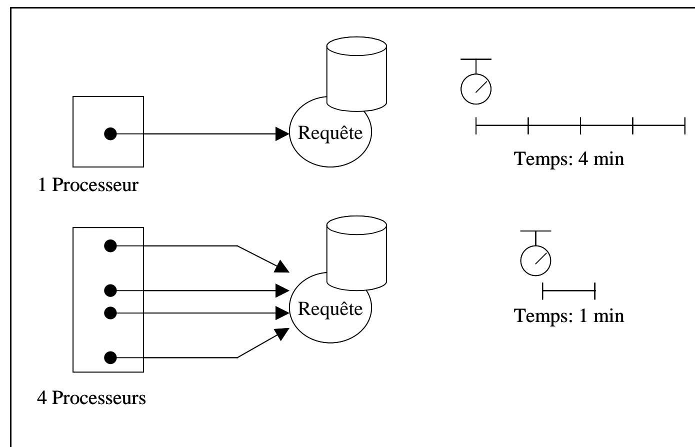


Figure 2-12 : Illustration du Facteur de Rapidité.

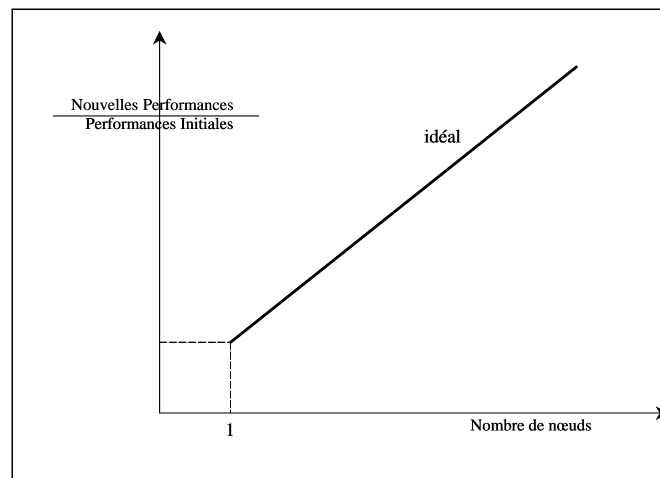


Figure 2-13: Courbe idéale du Facteur de Rapidité.

L'efficacité est égale au nombre de processeurs divisé par le facteur de rapidité obtenu. Si le rapport est égal à 1, l'efficacité est de 100%,. L'idéal est que la courbe du facteur de rapidité ait une pente de 45°, comme le montre la Figure 2-15. Cette courbe est rarement réalisée car le parallélisme induit des coûts supplémentaires.

Le *facteur de rapidité* ne tient compte de la configuration du système, le *facteur d'échelle* décrit ci-dessous tient compte à la fois de la taille du problème et de la configuration matérielle.

2.7.3 Facteur d'Echelle

Le facteur d'échelle ou facteur d'accroissement (ang. *scale-up*) mesure la conservation du temps de réponse en augmentant de façon proportionnelle la charge et le nombre de processeurs, par rapport à la configuration initiale.

$$Facteur\ \ d'échelle = \frac{Temps\ mis\ par\ le\ petit\ système\ pour\ exécuter\ le\ petit\ problème}{Temps\ mis\ par\ le\ gros\ système\ pour\ exécuter\ le\ gros\ problème}$$

La Figure 2-14 montre qu'une requête est exécutée en quatre minutes par un processeur, et quatre minutes sont aussi nécessaires pour l'exécution de quatre requêtes par quatre processeurs. Le facteur d'échelle est égal à 1.

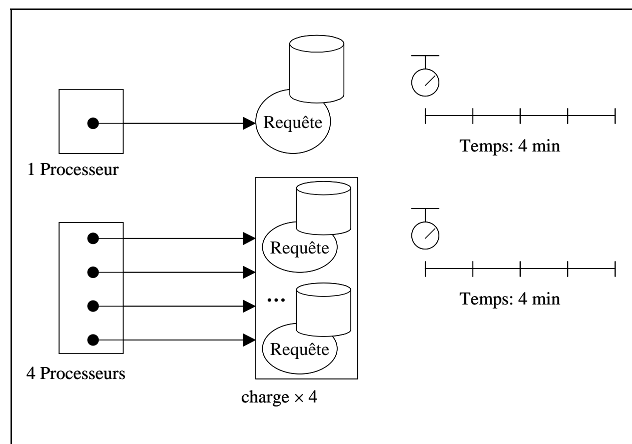


Figure 2-14 : Illustration du Facteur d'Echelle.

L'idéal est de conserver le même temps pour traiter une charge par un processeur, que *n* fois la charge par *n* processeurs. La courbe serait donc horizontale. En réalité, le temps de réponse augmente quand la charge augmente malgré l'ajout de processeurs.

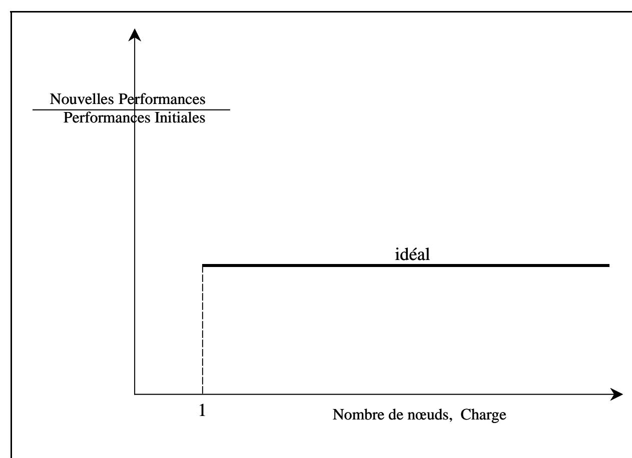


Figure 2-15 : Courbe idéale du Facteur d'Echelle.

Ainsi, découle la définition de la **scalabilité** (ang. *scalability*), comme étant la capacité d'une application de maintenir le même niveau de performances, quand la charge augmente et la configuration matérielle évolue -par l'ajout de processeurs et/ou de capacités de stockage.

2.7.4 Loi d'Amdhal

La loi d'Amdahl exprime l'amélioration des performances en fonction de α et P, tel que α désigne la fraction du programme devant être exécutée de façon séquentielle, par conséquent 1- α est la partie susceptible d'être améliorée ou parallélisable et P étant le facteur d'amélioration.

Gene Amdahl, a démontré que l'accélération maximale est limitée. En effet,

$$Accélération\ maximale \leq \frac{1}{\alpha + \frac{(1-\alpha)}{P}} \leq \frac{1}{\alpha}$$

Si le facteur d'amélioration est infini, l'amélioration résultante ne serait que de 1,67.

Cette loi est utile car elle montre que l'amélioration des performances par le parallélisme des tâches est limitée.

2.8 Conclusion

Tout au long de ce chapitre, nous avons évoqué des éléments fondamentaux pour la compréhension des systèmes distribués, et auxquels nous allons faire référence dans les chapitres suivants. Le chapitre suivant évoque le problème de la haute disponibilité, et énumère des mécanismes de haute disponibilité.

CHAPITRE

3 REVUE DE MECANISMES DE HAUTE DISPONIBILITE

3.1 Introduction

Pour assurer la haute disponibilité dans les systèmes de stockage distribués, la recherche s'est dirigée avec plus ou moins de succès dans plusieurs voies différentes. Certaines solutions proposées se sont avérées prohibitives, en terme de stockage ou de complexité de récupération de données perdues. Le présent chapitre survole et discute, à titre non exhaustif, des solutions de haute disponibilité évoquées dans la littérature [PGK88, HGK+94, B3M95, XB99, SS96, SS02, CEG+04, LMR98].

3.2 La Haute Disponibilité

Comme le montre la Table 1-1 (p.3), une interruption de service pendant un laps de temps induit des pertes désastreuses. Cette section évoque la problématique de la haute disponibilité, et présente une taxonomie des différentes solutions proposées dans la littérature, et énumère enfin des métriques d'évaluation d'une solution de haute disponibilité.

3.2.1 Classification de Systèmes à Haute Disponibilité

Gray et Siewiorek [GS91] proposent une classification des systèmes en fonction de leurs temps d'indisponibilité, sous l'hypothèse de continuité de service 24×7.

La classe de disponibilité d'un système est déduite du temps d'indisponibilité du système sur une échelle de temps d'un an. Dans la Table 3-1, remarquons que la classe de

disponibilité coïncide avec le nombre de '9's du pourcentage de disponibilité. Ainsi, la classe 2 désigne les systèmes disponibles 99% du temps, et la classe 5 désigne les systèmes disponibles 99,999% du temps.

Type du Système	*Indisponibilité (#minutes /an)*	*Disponibilité (%)*	*Classe de Disponibilité*
Non géré	50 000	90	1
Géré	5 000	99	2
Bien géré	500	99,9	3
Tolérant les fautes	50	99,99	4
Haute disponibilité	5	99,999	5
Très Haute disponibilité	0,5	99,9999	6
Ultra Haute disponibilité	0,05	99,99999	7

Table 3-1: Classification de systèmes en fonction du temps d'indisponibilité par an [GS91].

Pour rendre un système de stockage hautement disponibles, plusieurs stratégies existent. Wylie et al. [WBS+00, WBP+01, BWWG02] généralisent les solutions en *schéma seuil*-(*p*-*m*-*n*) (ang. *threshold scheme*) où, les données sont encodées en *n* blocs, tel que tout ensemble de *m* blocs parmi les *n* permet de reconstituer les données et au moins *p* blocs peuvent profiter à un intrus. Chaque schéma offre un niveau différent de performance, de disponibilité et de sécurité. Les auteurs énumèrent les schémas de haute disponibilité suivants :

(1) Schéma *n-way replication*: c'est un schéma seuil (1-1-*n*), où *n* répliquas des données source sont sauvés, tel qu'un répliqua permet de récupérer les données source et compromet la sécurité du système.

(2) Schéma de décimation : (ang. *Striping*) c'est un schéma seuil (1-*n*-*n*) où un bloc est subdivisé en *n* sous-blocs. Notons que *n* sous-blocs permettent de reconstruire les données source, et chaque sous-bloc compromet la sécurité du système.

(3) Schéma *Splitting*: c'est un schéma seuil (*n*-*n*-*n*) qui consiste à sauver *n*-1 valeurs aléatoires et une valeur calculée. Cette dernière est le XOR de la valeur originale et des *n*-1 valeurs. Dans ce schéma, *n* blocs permettent de reconstruire les données source et *n* blocs compromettent la sécurité du système.

(4) Schéma *Secret Sharing Schemes*: Shamir a proposé le schéma seuil *SS* (*m*, *m*, *n*) [S79]. Son implantation du 'Secret Sharing' est basée sur l'interpolation de points dans un polynôme d'un corps de Galois. Le polynôme de codage se compose d'une valeur secrète et de *m*-1 valeurs aléatoires.

(5) Schéma *Information Dispersal Algorithm*: M.O. Rabin a proposé le schéma IDA seuil (1-*m*-*n*) [R89] qui utilise la même méthode basée sur les polynômes mais sans utiliser des valeurs aléatoires, tel que *m* valeurs secrètes sont utilisées pour déterminer l'unique polynôme de codage.

(6) Schéma *Ramp*: c'est un schéma seuil (*p*-*m*-*n*) basé aussi sur les polynômes. Les points sont utilisés uniquement pour déterminer le polynôme de codage sont *p*-1 valeurs aléatoires et *m*-(*p*-1) valeurs secrètes. Pour *p* égal à 1, le schéma *Ramp* est

identique au schéma IDA et pour p égal à m le schéma *Ramp* est identique au schéma SS. Remarquons que la haute disponibilité et la sécurité des trois premiers schémas : réplication, décimation et éclatement dépendent seulement du paramètre n ; et que les aspects de sécurité se renforcent d'un schéma à un autre.

3.2.2 Fiabilité d'un Système

La fiabilité d'un système est un facteur quantifiable dont l'unité de mesure est le temps moyen entre deux pannes successives ou *MTBF* (ang. *Mean Time Between Failure*). Le *MTBF* mesure le nombre de fois où une machine tombe en panne durant une période donnée.

La disponibilité d'un système est également chiffrable sous forme de pourcentage calculé par l'expression suivante :

$$\text{Disponibilité du Système} = \frac{MTBF}{MTBF + MTTR}$$

Sachant que le paramètre *MTTR* (ang. *Mean Time To Repair*) représente le temps moyen de réparation d'une panne.

En supposant une non-corrélation des probabilités d'échecs, soit f_{noeud} la probabilité d'échec d'un nœud, la probabilité de récupération est égale à :

$$\text{Pr}\, obabilit\acute{e}\ de\ R\acute{e}cup\acute{e}ration = \sum_{i=0}^{n-m} C_i^n \cdot (f_{noeud})^i \cdot (1 - f_{noeud})^{n-i}$$

Bakkaloglu et al. [BWW02] ont défini un modèle où les échecs de nœuds sont en corrélation.

3.2.3 Critères d'Evaluation d'une Solution

D'après Gray et Siewiorek [GS91], les systèmes à haute disponibilité doivent :

(1) Tolérer les défaillances auxquelles ils sont sujets,

(2) Détecter et traiter les défaillances,

(3) Masquer les défaillances à l'utilisateur,

(4) Assurer une continuité de service même en cas d'échec,

(5) Garantir une remise en service rapide.

La comparaison de différentes solutions est ardue, car il entre en ligne de compte plusieurs critères, entre autres le coût de la solution, les performances, le temps de reprise, la résistance aux pannes, etc.

Hellerstein et al. [HGK+94] définissent des critères permettant d'évaluer une solution basée sur le codage/ décodage des données source. Nous supposons que les données de parité sont sauvées sur des volumes de parité, et les données source sont sauvées sur des volumes de données.

Coût de Stockage des Données de Parité (ou Taux de Codage) : C'est le ratio défini par le volume des données de parité divisé par le volume des données source.

Complexité (ou Pénalité) de mise à jour : C'est le nombre de volumes à mettre à jour suite à une mise à jour d'une donnée dans un volume de données.

Taille du Groupe : L'ensemble de volumes à accéder durant le processus de récupération constitue un groupe. C'est un critère important, car le temps de récupération est en fonction du nombre de volumes à accéder en cas d'échec. Ce facteur indique également le nombre de volumes opérationnels pour lesquels les performances d'accès sont en mode dégradé, puisque ces volumes sont exploités par le processus de récupération.

Expansion du système de stockage : L'ajout de nouveaux volumes de données ou de parité au système de stockage devrait s'effectuer sans le re-calcul du contenu des volumes de parité existants.

Complexité de codage et de décodage : La complexité de codage/ décodage s'exprime en premier lieu en fonction du nombre d'opérations arithmétiques de base, et en second lieu en secondes (temps de codage/décodage).

3.2.4 Codes de Correction d'Effacements versus Réplication

Plusieurs études de mesure de la fiabilité ont été conduites afin de comparer les systèmes tolérant l'échec par l'utilisation de codes correcteurs d'effacements (ang. *erasure resilient codes*) aux méthodes de réplication. Nous citons celles de Schwarz [S02] et Weathersoopn et Kubiatowicz [WK02], qui ont démontré que par rapport aux codes correcteurs d'effacements, les stratégies de réplication sont pénalisantes au vu des critères de stockage et de transfert de données (bande passante).

3.3 Réplication

Plusieurs schémas de réplication existent. Les principaux paramètres d'un schéma de réplication sont le nombre de répliquas et le mode de rafraîchissement des répliquas. Dans ce qui suit, nous énumérons à titre non exhaustif quelques travaux utilisant la réplication pour augmenter la haute disponibilité des systèmes de stockage de données.

Notons que le principal avantage des systèmes à base de réplication est que les miroirs sont fonctionnels et peuvent être interrogés afin de diminuer la charge du site maître.

3.3.1 Schéma de Réplication *SSS*

Schlude, Soininen et Widmayer [SSW02] proposent une structure de données distribuée à haute disponibilité basée sur les arbres binaires. Les nœuds feuilles sauvent les articles, alors que les nœuds internes constituent des routeurs vers les nœuds feuilles. Chaque nœud u est muni d'un intervalle de responsabilité: I_u égal à $[l_u, r_u]$. Pour chaque nœud u

existe un nœud miroir *u*' de *u*. Ainsi, en cas d'échec de *u*, les données sont récupérées de *u*'.

Chaque client a une image du degré de bifurcation et de la profondeur de chaque arbre. L'image du client n'est pas nécessairement correcte, et est mise à jour chaque fois que le client commet une erreur d'adressage.

Le fichier s'agrandit à travers les éclatements. En effet, un serveur surchargé *u* éclate en deux serveurs : *v1* et *v2*, qui deviennent des nœuds feuilles, alors que *u* devient un nœud interne. La moitié du contenu de *u* est transférée à *v1*, et l'autre moitié migre vers un nouveau serveur qui hébergera *v2*. L'article propose une stratégie d'allocation à la fois économique de point de vue stockage et astucieuse. Le nouveau serveur hébergera le fils de droite de *u* et le fils de gauche de *u'*, comme le montre la Figure 3-1.

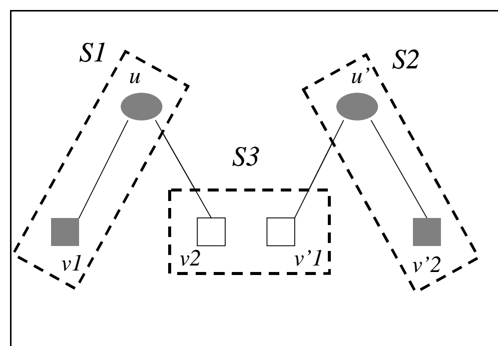


Figure 3-1 : Eclatement de deux nœuds *u* et *u'*, et création d'un nouveau serveur *S3*.

La haute disponibilité n'est pas transparente aux sites client. Par conséquent, toute manipulation client (insertion, suppression, mise à jour) est envoyée aux deux structures. Mis à part les messages d'acheminement des requêtes, d'autres messages sont nécessaires notamment ceux pour:

- Synchroniser les éclatements entre les deux arbres pour que les nœuds qui se correspondent, aient les mêmes intervalles de responsabilité.
- Contourner les serveurs en échec pour acheminer les requêtes. Pour ce, chaque nœud transfère la requête client à son fils et au nœud miroir de son fils. Le non-double transfert de la requête est assuré par une liste traçant les derniers messages reçus. Une requête sauvée dans la liste est supprimée dès la réception de la requête duplicata.

Le schéma de haute disponibilité proposé est contraint par :

- Le nombre de messages de coordination échangés entre les deux arbres. Les scénarios sont conçus partant du fait de la difficulté de mise en œuvre d'un consensus dans un milieu distribué sensible aux échecs.
- La complexité de récupération, résultant du fait que l'échec d'un serveur implique la reconstruction de tous les nœuds qu'il héberge nœuds internes ou feuilles soient-ils.

– Le nombre de serveurs requis, que nous estimons à $2^{n-1}+1\left(2+\sum_{i=2}^{n}2^{i-2}\right)$, pour un arbre de profondeur n.

3.3.2 Schémas de Réplication Optimisés

E. K. Lee s'intéresse dans cet article [L95] au problème d'équilibre de charge dans un système de stockage en échec, où la haute disponibilité est assurée par la réplication. Le schéma de haute disponibilité basée sur les miroirs présente un problème de déséquilibre de charge en cas d'échec d'un serveur. En effet, le serveur miroir du serveur en échec assumerait la charge du serveur en échec, alors que les autres serveurs du système ne supportent aucune extra charge. E. K. Lee propose des schémas de placement de données pour répartir la charge entre l'ensemble des serveurs en cas d'échec d'un serveur. Notamment, le schéma *Chained-Declustering*, et le schéma *Multi-Chained-Declustering*.

3.3.2.1 Schéma *Chained-Declustering*

Supposons disposer de m serveurs, l'unité de données U_x est hébergée par le serveur $S_{x \bmod m}$. L'unité U'_x (réplica de U_x) est hébergée par le successeur du serveur $S_{x \bmod m}$ (sachant que le successeur du serveur S_{m-1} est le serveur S_0). Ainsi, en cas d'échec d'un serveur S_i, le contenu d'un serveur S_i est accessible à partir du *Successeur* de S_i et du *Prédécesseur* de S_i. Ce schéma réalise une répartition de la charge de S_i entre seulement deux serveurs. Pour plus de performances, le système déchargerait les serveurs *Successeur*(S_i) et *Prédécesseur*(S_i) participants à la récupération, en ré-orientant les requêtes vers leurs miroirs respectifs.

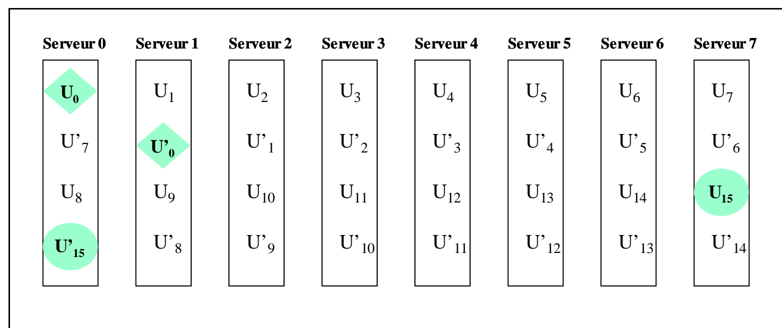


Figure 3-2: Schéma *Chained-Declustering* pour *m* = 8.

3.3.2.2 Schéma *Multi-Chained-Declustering*

Supposons disposer de m serveurs, l'unité de données U_x est hébergée par le serveur $S_{x \bmod m}$. L'unité U'_x est hébergée par le *pas-Successeur* du serveur $S_{x \bmod m}$. Par exemple, le *2-Sucesseur*(S_{m-2}) est le serveur S_0. E. K. Lee n'a pas précisé, si dans le schéma *Multi-Chained-Declustering*, toutes les m unités le *pas* est incrémenté, ou reste dans l'ensemble {1, 2}. Dans le premier cas, si U_x est affecté au serveur $S_{x \bmod m}$ alors U'_x serait affecté au serveur *(1+(x mod m))-Sucesseur*($S_{x \bmod m}$). Notons qu'en cas d'échec d'un serveur S_i, le contenu d'un serveur S_i est accessible à partir de 2^{pas} serveurs. Ainsi, le schéma réalise une répartition de la charge entre 2^{pas} serveurs.

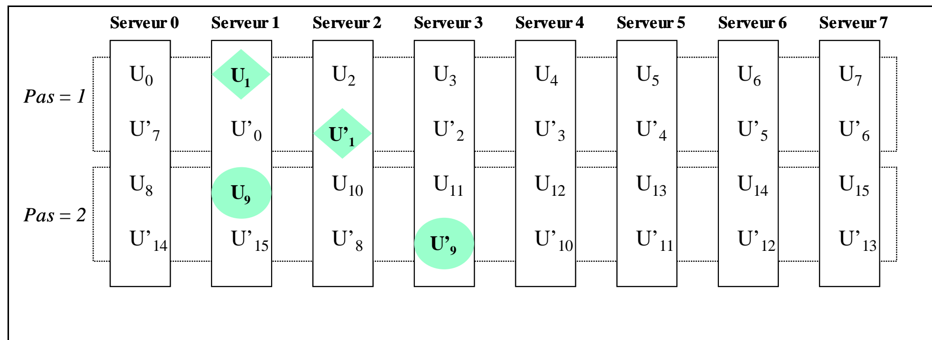


Figure 3-3: Schéma Multi-Chained-Declustering pour *m* = 8.

3.4 La Technologie RAID

Les disques RAID (acronyme de *Redundant Arrays of Independent Disks*) ont été conçus en 1987 par trois chercheurs de l'université de Berkeley D.A. Patterson, G. Gibson et R.H. Katz [PGK88].

Un système RAID est une matrice de disques dans laquelle une partie de la capacité physique est utilisée pour stocker de l'information redondante. Le système d'exploitation et les applications voient la matrice de disques comme étant un seul disque. Un système RAID permet de :

* Augmenter la capacité : RAID permet de mettre «bout à bout » des disques durs, ce qui permet d'accroître le volume de stockage.
* Améliorer les performances : les données sont écrites sur plusieurs disques à la fois. Ainsi, chacun des disques n'a qu'une partie de données à inscrire.
* Assurer la tolérance de panne : le système RAID permet de se prémunir contre les défaillances d'un disque.

Dans l'article original [PGK88], une taxonomie de cinq architectures fut proposée. Toutefois, il faut signaler qu'il existe d'innombrables architectures de disques RAID. Toutes les architectures de disques RAID sont en fait une combinaison des deux facteurs suivants :

* La manière dont les informations de redondance sont distribuées sur l'ensemble des disques.
* Le type de redondance utilisé (réplication, parité).

Il y'a deux façons de mettre en œuvre le RAID : soit niveau logiciel (*software*) géré par le système d'exploitation ou niveau matériel (*hardware*) [F96][003].

RAID niveau 1: RAID niveau 1 ou le mode miroir est l'option la plus coûteuse en terme de stockage, puisque tous les disques sont dupliqués. Cette duplication de disques entraîne une multiplication par deux du coût du système et une utilisation effective de seulement 50% de la capacité de stockage. Chaque mise à jour est faite sur les deux disques et quand le disque principal échoue, les utilisateurs basculent automatiquement au miroir. Les performances en lecture sont bonnes car pour les requêtes on peut utiliser,

aussi bien les disques d'information et les disques miroirs. Ce qui permet de distribuer la charge sur l'ensemble des disques.

RAID niveau 2 : Ce niveau se distingue par la présence de disques parité, dont le nombre est proportionnel au nombre de disques de données. Ces extra-disques, générés par les codes de Hamming, permettent de récupérer les disques en échec. Le RAID 2 est très peu utilisé, et peu documenté.

RAID niveau 3: Ce niveau ne comporte qu'un seul disque de parité généré par la parité impaire (ou-exclusif ou XOR). Dans ce niveau les données sont distribuées bit par bit sur l'ensemble des disques de données. Cette répartition pénalise la performance des transactions d'E/S. En effet, la reconstitution d'une unité de transfert nécessite plusieurs E/Ss à différents disques. Chaque écriture sur un disque de données entraîne une écriture sur le disque de parité, ce dernier devient rapidement un goulot d'étranglement.

RAID niveau 4 : Le niveau 4 utilise une granularité plus forte en ce qui concerne la taille des données réparties. En effet, contrairement au niveau 3, le niveau 4 n'entrelace pas les données sur plusieurs disques de bit en bit, mais de bloc en bloc (un bloc peut être une piste, un secteur ou un cylindre). Cette architecture améliore les performances des transactions d'Entrées/ Sorties. Néanmoins, si les accès en lecture ont un taux de parallélisation très fort, il en n'est pas de même pour les écritures. En effet, un accès en écriture nécessite la lecture de deux blocs et l'écriture de deux blocs. Avant l'écriture, on lit le bloc de données à mettre à jour ainsi que le bloc de parité, puis on écrit le bloc de données, enfin on effectue le calcul de parité afin de mettre à jour le bloc de parité. Ces quatre accès pour une transaction d'écriture constitue un goulot d'étranglement des écritures sur le disque de parité. Le problème de goulot d'étranglement observé sera adressé dans le niveau cinq.

RAID niveau 5 : Le niveau cinq résout le problème de goulot d'étranglement sur le disque de parité, en distribuant les blocs de parité sur l'ensemble des disques. Cette architecture est la plus répondue dans l'industrie de stockage.

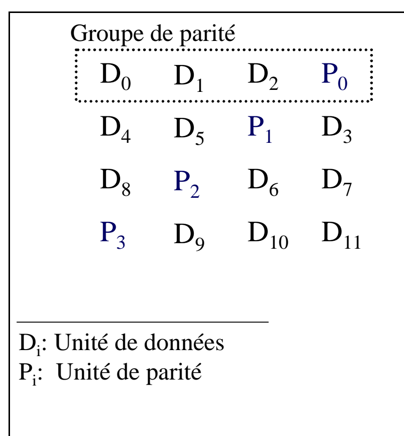


Figure 3-4 : RAID niveau 5.

Combinaisons de Niveaux RAID : Mogi et al., [MK96] proposent de combiner les niveaux 1 et 5. Ils proposent de diviser l'espace de stockage en deux parties : une partie miroir – RAID niveau 1 contenant les données correspondant à une importante fréquence d'accès, et une autre partie niveau 5 contenant le reste des données. Un processus de migration des données est requis puisque le taux d'accès aux données évolue dans le temps.

3.5 Codes Linéaires Binaires

Hellerstein et al. [HGK+94] proposent des codes surmontant t effacements avec une restriction sur t, où $t \leq 5$. Dans ce qui suit, nous définissons les codes linéaires binaires, puis nous dérivons les codes énumérés dans l'article, correspondant à des valeurs particulières de t : 1, 2, 3.

3.5.1 Codage/ Décodage

Processus de Codage

Un code linéaire binaire est défini par une matrice de parité H $(c \times (k+c))$, où c dénote le nombre de disques de parité et k le nombre de disques de données. La matrice H est la concaténation de $P(c \times k)$ – la matrice qui détermine les équations de codage et $I(c\times c)$ la matrice d'identité. Remarquons que chaque ligne de H représente un groupe de parité, les disques y appartenant sont marqués par des '1's.

Processus de Décodage

La récupération n'est possible que si toutes c colonnes de H sont linéairement indépendantes. Or, un ensemble de vecteurs binaires est linéairement indépendant si et seulement si le vecteur somme est différent du vecteur zéro. Désignons par X le vecteur contenant un symbole de chaque disque, $H*X = 0$.

Si f disques sont en échec, les colonnes de H et les entrées de X sont divisées en deux types, qui sont : celles qui représentent des disques en échec et celles qui représentent les disques survivants. H est égal à $[A|B]$ et X est égal à $[d \mid y]$, où y est le vecteur des entrées à récupérer, et B les colonnes des disques en échec. Le processus de décodage se réduit à la détermination des entrées du vecteur y par la résolution du système linéaire $H*X = A*d + B*y = 0$.

Remarquons que $B*y$ constitue un ensemble de c équations de décodage, alors que f équations suffisent pour résoudre notre système. Nous désignons alors par B' : les f lignes de B devant être linéairement indépendantes ; et par A', les f lignes correspondantes dans A. Notre système devient $A'*d + B'*y = 0$, le vecteur y est égal à $(B')^{-1}*A'*d$. En ce qui concerne le choix des f lignes, chacune de ces dernières doit marquer au moins un '1' avec une colonne correspondant à un disque en échec.

Remarquons que dans le cas particulier d'un seul échec, B' serait une ligne de B impliquant l'ensemble des disques appartenant à ce groupe de parité. La ligne sélectionnée doit marquer un '1' avec la colonne correspondant au disque en échec.

3.5.2 Le code *1d-parity*

Le code consiste à diviser l'ensemble des disques de données en groupes de tailles *m*, et associer à chaque groupe un disque de parité.

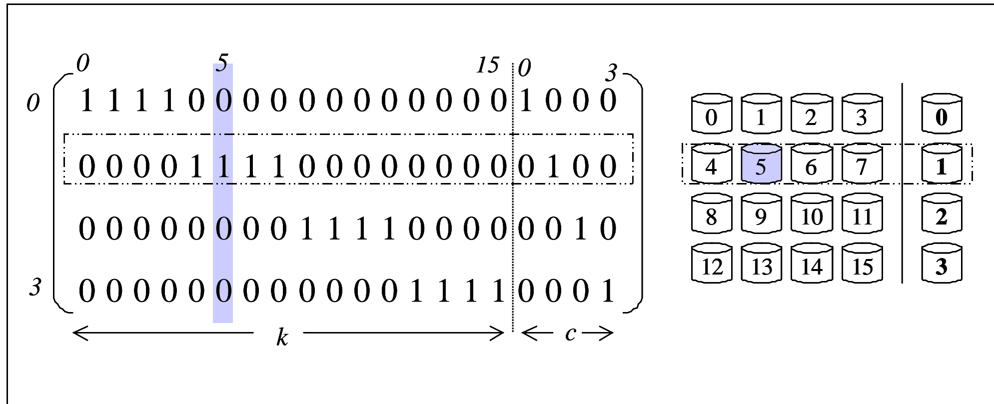


Figure 3-5 : Illustration du schéma *1d-parity* pour (k = 16, c = 4).

La matrice de parité H correspondant à ce code est telle que chaque colonne j de H a un poids égal à 1 ($\underset{\substack{i:0..c-1 \\ j:0..k+c-1}}{H[i][j]} = 1$ ssi le '1' se trouve sur la ligne i tel que $i = j \bmod c$).

La Figure 3-5 montre (à gauche) la matrice de parité correspondant à k = 16 et c = 4 et (à droite) la disposition des disques où les disques de parité sont numérotés en gras.

Le volume des données de parité représente le un-$m^{éme}$: $\frac{c}{k} = \frac{1}{m}$ du volume des données source.

3.5.3 Les Codes Correcteurs à 2 Effacements

L'article évoque deux codes, le *code full-2* et le *code 2d-parity*. Les deux codes corrigent tout échec affectant un couple de disques, et un ensemble d'échecs de trois disques. Dans le dernier cas, il ne faut pas que le disque de données et les deux disques de parités qui lui sont associés soient en échec.

3.5.3.1 Le Code *Full-2*

Le full-2 code est défini par une matrice de parité $H_{full2} = [P_{full2} \mid I]$, où P_{full2} est l'ensemble des différentes colonnes de poids 2. Le nombre de colonnes k possibles est égal à $C_c^2 = \frac{c*(c-1)}{2}$.

3.5.3.2 Le Code *2d-parity*

Désignons par m la taille d'un groupe, il s'agit de placer m^2 disques de données dans un tableau à 2 dimensions. Un disque de données appartient à deux groupes de parité, selon sa ligne et selon sa colonne.

La matrice de parité H correspondant à ce code est telle que le poids de chaque colonne j est égal à 2. Le premier '1' se trouve sur la ligne i tel que $(i \leftarrow j \bmod c/2)$. Le deuxième '1' se trouve sur la ligne i tel que $(i \leftarrow i')$ si $(i' \geq c/2)$, sinon $(i \leftarrow i' + c/2)$, avec $(i' = j + \lfloor 2*j/c \rfloor \bmod c)$.

La Figure 3-6 montre (à gauche) la matrice de parité correspondant à k = 16 et c = 4 et (à droite) la disposition des disques où les disques de parité sont numérotés en gras.

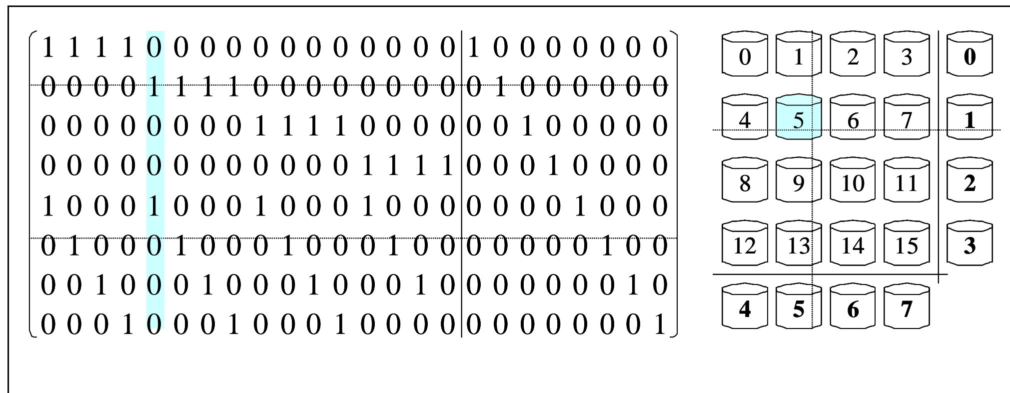


Figure 3-6 : Illustration du *2d-parity* pour (k = 16, c = 4).

3.5.3.3 Comparaison

Dans ce qui suit, nous dressons un tableau comparant les deux codes par rapport à k le nombre de disques de données et $\frac{c}{k}$ le taux de codage, et ce en fixant la variable c -étant le nombre de disques de parité employés.

	Code *2d-parity*	Code *full-2*
k	$\frac{c^2}{4}$	$\frac{c*(c-1)}{2}$
$\frac{c}{k}$	$\frac{4}{c}$	$\frac{2}{(c-1)}$

Table 3-2 : Comparaison des codes *2d-parity* et *full-2*.

Remarquons que, le *code full-2* emploie $\frac{c^2 - 2*c}{4}$ disques de données plus que le *code 2d-parity*. Ceci implique que pour le même nombre c de disques de parité, le *code 2d-parity* s'avère plus avantageux que le *code full-2* puisque son taux de codage est inférieur.

En fixant le nombre de disques de données k, le *code 2d-parity* emploie $2\sqrt{k} - \frac{(1+\sqrt{1+8k})}{2}$ disques de parité plus que le *code full-2*.

3.5.4 Les Codes Correcteurs de 3 Effacements

L'article évoque quatre codes, le *code full-3*, le *code 3d-parity*, le *code Steiner* et le *code additive-3*. Dans ce qui suit, nous les décrivons.

3.5.4.1 Le Code *Full-3*

Le *code full-3* est défini par une matrice de parité $H_{full3} = [P_{full3}| I]$, où P_{full3} est l'ensemble des différentes colonnes de poids 3. Le nombre de colonnes k est égal à $C_c^3 = \frac{c*(c-1)*(c-2)}{6}$.

3.5.4.2 Le Code *3d-parity*

Désignons par m la taille d'un groupe, il s'agit de placer m^3 disques de données dans un tableau à 3 dimensions. Un disque de données appartient à trois groupes de parité, selon trois axes.

3.5.4.3 Le Code *Steiner*

Le *code Steiner* est défini seulement pour des valeurs de c puissances de 3. Le code est défini par une matrice de parité $H_{Steiner} = [P_{Steiner}|I]$. La matrice $P_{Steiner}$ se compose des colonnes qui contiennent des '1's aux lignes q, r et s, et tel que l'ensemble $\{q, r, s\}$ est sous-ensemble de Steiner(c).

Steiner(c) est défini de manière récursive, comme suit,

(1) Steiner(3) = {{0, 1, 2}}.

(2) Pour $c > 3$,

$$Steiner(c) = Steiner(\frac{c}{3}) \cup (\frac{c}{3} + Steiner(\frac{c}{3})) \cup (\frac{2c}{3} + Steiner(\frac{c}{3}))$$

$$\cup \{ \{i, \frac{c}{3} + ((i+j) \bmod \frac{c}{3}), \frac{2c}{3} + ((i+2j) \bmod \frac{c}{3})\},$$

Tels que $i, j \in \{0, 1, \ldots, \frac{c}{3}-1\}$ }.

k + *Steiner*(c') est l'ensemble obtenu en ajoutant k à chaque élément du sous-ensemble de *Steiner*(c').

<u>*Exemple:*</u> Détermination de Steiner(9)

$$Steiner(9) = Steiner(3) \cup 3+ Steiner(3) \cup 6+ Steiner(3) \cup$$
$$\{ \{i, 3+ ((i+j) \bmod 3), 6+((i+2j) \bmod 3)\}, \text{tels que } i, j \in \{0, 1, 2\} \}.$$
$$= \{ \{0, 1, 2\}, \{3, 4, 5\}, \{6, 7, 8\}, \{0, 3, 6\}, \{0, 4, 8\}, \{0, 5, 7\}, \{1, 4, 7\}, \{1, 5, 6\}, \{1, 3, 8\}, \{2, 5, 8\}, \{2, 3, 7\}, \{2, 4, 6\} \}$$

3.5.4.4 Le Code *Additive-3*

Ce code est défini pour des valeurs de c multiples de 3. La matrice de parité correspondante est $H_{additive-3} = [P_{additive-3} \mid I]$. $P_{additive-3}$ se compose de toutes les colonnes de poids 3, contenant des ‘1’s aux lignes q, r et s, tel que ($q + r + s \ mod\ c = 1$).

Il existe $c^2 - 3c$: ($C_c^1 * C_c^1 * C_1^1$ - $3 * C_c^1 * C_1^1 * C_1^1$) manières d’affecter des valeurs à q, r et s, satisfaisant l’équation ($q + r + s \ mod\ c = 1$). En divisant par 3!, le nombre de permutations possibles de q, r et s; nous obtenons le nombre de disques de données k.

3.5.4.5 Comparaison

Dans ce qui suit, nous dressons un tableau comparant les quatre codes par rapport à k le nombre de disques de données et $\frac{c}{k}$ le taux de codage, en fixant la variable c -étant le nombre de disques de parité employés.

	Code 3d-parity	*Code full-3*	*Code Steiner*	*Code additive-3*
k	$\frac{c^3}{27}$	$\frac{c*(c-1)*(c-2)}{6}$	$\frac{c*(c-1)}{6}$	$\frac{c^2-3c}{6}$
$\frac{c}{k}$	$\frac{27}{c^2}$	$\frac{6}{(c-1)*(c-2)}$	$\frac{6}{(c-1)}$	$\frac{6}{(c-3)}$

Table 3-3 : Comparaison des codes *3d-parity, full-3, Steiner* et *additive-3*.

3.5.5 Conclusion

Il est intéressant de généraliser à t>3 les résultats. Remarquons que la conception devient plus complexe. En effet, dans la matrice de parité H, où P consiste de différentes colonnes de poids t, le code n’est pas forcément correcteur de t-effacements, car le vecteur somme peut être nul. Dans ce qui suit, nous décrivons des schémas de parité 2-disponibles : EVENODD [B3M95] (§3.6), X-code [XB99-a, XB99-b, X98] (§3.7), RDP [CEG+04] (§3.9), plus économiques de point de vue taux de codage que le *full-2* et le *2d-parity*.

3.6 Schéma EVENODD

Blaum, et al. [B3M95] proposent un schéma 2-disponible dont le calcul de parité et la récupération se basent sur le ou-exclusif (XOR). Le schéma de parité calcule deux disques de parité à partir de m disques de données, et survit à l’échec de deux disques. Dans ce qui suit, nous énonçons et illustrons par les processus de codage/décodage de EVENODD.

3.6.1 Processus de Codage

Pour simplifier, nous supposons qu'un disque contient *m-1* symboles (un symbole peut être un bit, un octet etc.). Le schéma EVENODD calcule deux disques de parité pour un groupe de *m* disques de données. Les symboles des disques de parité sont calculés selon l'Algorithme 3-1. Notons que, le premier disque de parité est la parité impaire horizontale (pente 0), alors que le deuxième disque de parité est la parité impaire selon la diagonale de pente –1.

L'ensemble des symboles de données et de parité sont placés dans un tableau à 2-dimensions $(m-1) * (m+2)$ (voir exemple $m = 5$, Figure 3-7).

Algorithme 3-1: Codage dans le schéma EVENODD.

Le premier disque de parité

$$\forall\, l \in [0, m-2], \quad a_{l,m} = \sum_{t=0}^{m-1} a_{l,t}$$

Tel que, $a_{l,m}$ est le $l^{ième}$ symbole du disque m est la parité impaire des $l^{èmes}$ symboles de chaque disque d'information.

Le deuxième disque de parité

$$\begin{cases} \forall\, l \in [0, m-2], \quad a_{l,m} = S \oplus \left(\sum_{t=0}^{m-1} a_{\langle l-t \rangle_m, t} \right) \\ S = \sum_{t=1}^{m-1} a_{m-1-t, t} \end{cases}$$

$$\langle x \rangle_y \leftrightarrow x \bmod y$$

Complexité du Codage

Il est intéressant de déterminer la complexité de calcul des symboles de chacun des deux disques de parité, exprimée en nombre d'opérations XOR de symboles. Pour le 1er disque de parité, le calcul d'un symbole de parité nécessite (m-1) XORs. Effectivement, c'est le nombre de disques de données m-1. Pour le 2éme disque de parité, S nécessite (m-2) opérations XOR. Chaque symbole du disque $m+1$ se calcule en m opérations XOR.

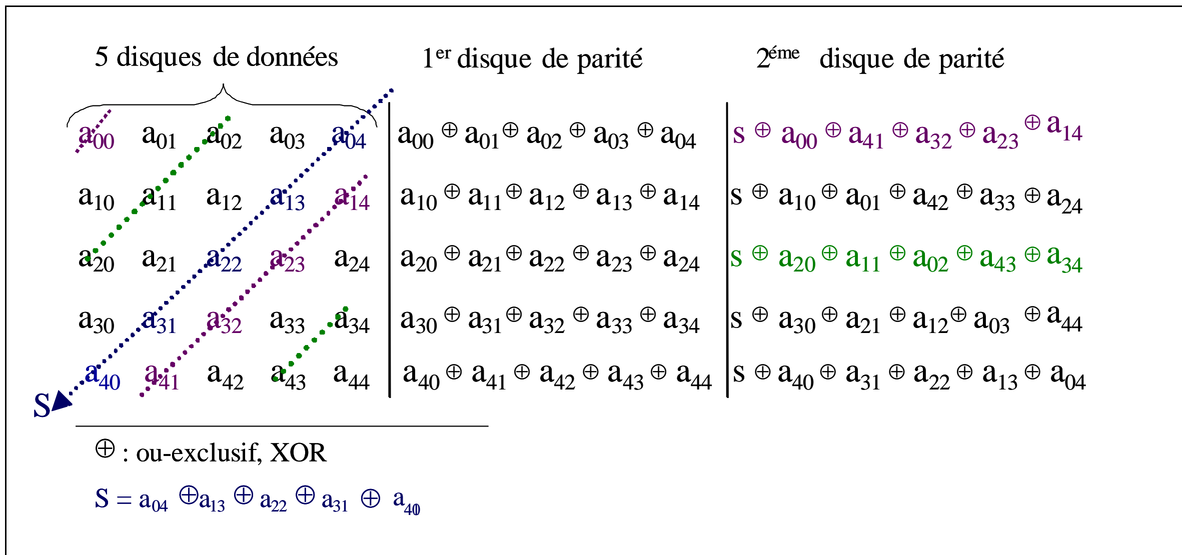


Figure 3-7 : EVENODD, pour *m* = 5.

3.6.2 Processus de Décodage

L'algorithme proposé, et illustré ci-suit, ne permet de prévenir que contre l'échec d'un disque ou de deux disques seulement.

Algorithme 3-2 : Décodage dans le schéma EVENODD.

<u>1er Cas: Echec d'un seul disque</u>

* *Cas 1.1* : Echec d'un disque de données ou du 1er disque de parité,

Le contenu du disque en échec est le XOR de *m* disques disponibles sauf le disque *m+1*.

* *Cas 1.2* : Echec du 2éme disque de parité,

Il s'agit de régénérer le contenu du disque *m+1*.

<u>2éme Cas: Echec de deux disques</u>

* *Cas 2.1* : Echec des deux disques de parité,

Exécution de l'algo. de codage (Algorithme 3-1).

* *Cas 2.2* : Echec d'un disque de données : i ($i < m$) et du disque de parité : $m+1$

Nous procédons en deux étapes. Dans une 1ére étape, nous récupérons le contenu du disque i à partir des $m-1$ disques de données disponibles et du disque m. Puis, nous recodons pour récupérer le disque $m+1$.

* *Cas 2.3* : Echec d'un disque de données : i ($i < m$) et du disque de parité : m

Nous commençons par récupérer le contenu du disque i. Puis nous recodons pour récupérer le disque m, étant le XOR des disques de données. Pour récupérer le disque i, il faut calculer S. Pour ce, nous considérons l'équation incluant le symbole $a_{m-1,i}$. Ce dernier est l'intersection de la colonne correspondant au disque i et de la $m-1^{éme}$ ligne. Etant donné que la $m-1^{iéme}$ ligne est une ligne factice, les symboles $a_{m-1,i}$ sont nuls. L'équation suivante détermine le symbole

$a_{<i-1>m, m+1}$ appartenant au disque $m+1$.

Pour $t \in [0, m\text{-}1]$, $a_{<i-1>m, m+1} = S + \Sigma a_{<i-1-t>m, t}$

➔ $S = (\Sigma a_{<i-1-t>m, t}$, pour $t \in [0, m\text{-}1]) + a_{<i-1>m, m+1}$.

* *Cas 2.4* : Echec de deux disques de données : i ($i < m$) et j ($j < m$)

Nous commençons par calculer S. D'après les équations déterminant les symboles de parité du disque m, nous avons :

$a_{0,m} = a_{0,0} \oplus a_{0,1} \oplus a_{0,2} \oplus \ldots \oplus a_{0,m-1}$

…

$a_{m-1,m} = a_{m-1,0} \oplus a_{m-1,1} \oplus a_{m-1,2} \oplus \ldots \oplus a_{m-1,m-1}$

➔Pour $l \in [0, m\text{-}1]$, $\Sigma a_{l,m} = \Sigma a_{l,0} \oplus \Sigma a_{l,1} \oplus \Sigma a_{l,2} \oplus \ldots \oplus \Sigma a_{l,m-1}$

D'après les équations déterminant les symboles de parité du disque m, nous avons :

$a_{0,m+1} = S \oplus a_{0,0} \oplus a_{m-1,1} \oplus a_{m-2,2} \oplus \ldots \oplus a_{1,m-1}$

…

$a_{m-2,m+1} = S \oplus a_{m-2,0} \oplus a_{m-3,1} \oplus a_{m-4,2} \oplus \ldots \oplus a_{m-1,m-1}$

$a_{m-1,m+1} = S \oplus a_{m-1,0} \oplus a_{m-2,1} \oplus a_{m-3,2} \oplus \ldots \oplus a_{0,m-1}$

Sachant que m est impair,

Pour $l \in [0, m\text{-}1]$, $\Sigma a_{l,m+1} = S \oplus \underline{\Sigma a_{l,0} \oplus \Sigma a_{l,1} \oplus \Sigma a_{l,2} \oplus \ldots \oplus \Sigma a_{l,m-1}}$.

Notons que le terme souligné n'est autre que $\Sigma a_{l,m}$, et ce d'après l'égalité démontrée ci dessous. Ceci nous permet d'exprimer S en fonction des symboles de parité.

$$S = \Sigma a_{l,m} \oplus \Sigma a_{l,m+1}, \text{ pour } l \in [0, m-1].$$

Une fois la valeur de S calculée, la récupération des disques i et j, revient à résoudre un problème d'équations linéaires, au nombre de $2*m$, et à $2*m$ inconnues. Les dernières étant les symboles des disques i et j. Notons enfin que les symboles $a_{m-1,i}$ et $a_{m-1,j}$ sont nuls, puisqu'ils appartiennent à la ligne factice.

3.6.3 Discussion

Vu que l'emploi de disques de parité dédiés est susceptible de dégrader les performances, cette approche peut être étendue au RAID niveau 5, et ce en distribuant les symboles de parité sur les $m+2$ disques.

Notons qu'une opération de mise à jour d'un symbole d'un disque de données, se répercute sur les deux symboles de parité. Toutefois, une mise à jour d'un symbole de la diagonale S a pour conséquence la mise à jour, de tous les symboles du deuxième disque de parité. Ceci est problématique et c'est ainsi que le *code X* a été proposé.

3.7 Schéma X-Code

Xu et Bruck [XB99-a, XB99-b, X98] proposent un nouveau code, *X-code*. Le code permet de prévenir contre deux échecs. Sa complexité de mise à jour, c'est à dire le

nombre de symboles de parité mis à jour suite à une mise à jour d'un symbole de données, est exactement égale à 2, et ce contrairement au schéma EVENODD [B3M95]. La deuxième particularité du code est le fait que les symboles de parité et de données coexistent dans les mêmes colonnes. Ces dernières représentent des volumes de données Ceci réduit la concurrence d'accès en mise à jour des symboles de parité, et ce contrairement au cas où des unités de stockage sont dédiées aux données de parité.

Dans ce qui suit, nous énonçons et illustrons par un exemple chacun des algorithmes de codage et de décodage.

3.7.1 Processus de Codage

Dans *X-code*, les symboles de données sont placés dans un tableau *(m-2)*m* et les symboles de parité sont sur les deux dernières lignes et sont calculées par l'Algorithme 3-3. La taille du code est $m*m$.

Algorithme 3-3 : Equations calculant les lignes *m*-1 et *m*.

$$\forall\, i \in [0, m-1],\quad \begin{cases} a_{m-2,\,i} = \sum_{t=0}^{m-3} a_{t,\,<i+t+2>_m} \\ a_{m-1,\,i} = \sum_{t=0}^{m-3} a_{t,\,<i-t-2>_m} \end{cases}$$

$$< x >_y \;\leftrightarrow\; x \bmod y$$

La Figure 3-8 illustre selon les règles de codage pour $m = 5$. Géométriquement parlant, les règles de codage sont selon les diagonales de pentes 1 et –1.

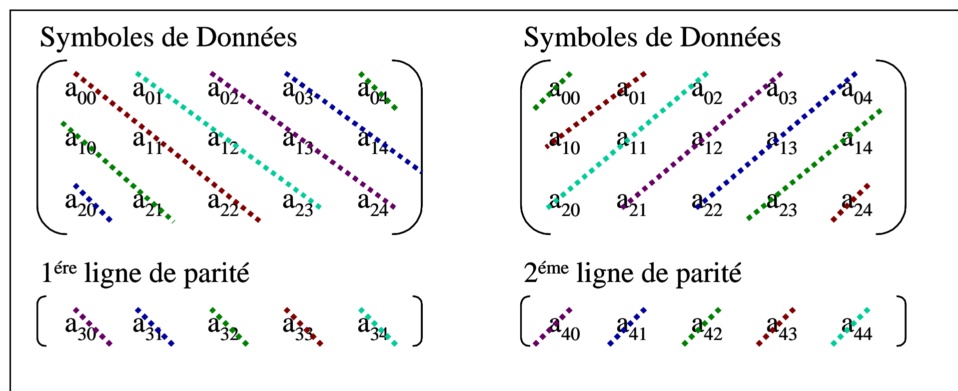


Figure 3-8 : Illustration de X-Code (*m* = 5).

Dans ce qui suit, nous étudions de prés selon notre propre interprétation le processus de codage, pour voir pour chaque symbole de parité quels sont les symboles de données desquels il est calculé.

$a_{m-2,0} = a_{0,2} \oplus a_{1,3} \oplus \ldots \oplus a_{m-3,m-1}$ $a_{m-2,1} = a_{0,3} \oplus a_{1,4} \oplus \ldots \oplus a_{m-3,0}$ $a_{m-2,2} = a_{0,4} \oplus a_{1,5} \oplus \ldots \oplus a_{m-3,1}$ … $a_{m-2,i} = a_{0,i+2} \oplus a_{1,i+3} \oplus \ldots \oplus a_{m-3,i-1}$ … $a_{m-2,m-1} = a_{0,1} \oplus a_{1,2} \oplus \ldots \oplus a_{m-3,m-2}$	$a_{m-1,0} = a_{0,m-2} \oplus a_{1,m-3} \oplus \ldots \oplus a_{m-3,1}$ $a_{m-1,1} = a_{0,m-1} \oplus a_{1,m-2} \oplus \ldots \oplus a_{m-3,2}$ $a_{m-1,2} = a_{0,0} \oplus a_{1,m-1} \oplus \ldots \oplus a_{m-3,3}$ … $a_{m-1,i} = a_{0,i-2} \oplus a_{1,i-3} \oplus \ldots \oplus a_{m-3,i+1}$ … $a_{m-1,m-1} = a_{0,m-3} \oplus a_{1,m-4} \oplus \ldots \oplus a_{m-3,0}$
(*a*) 1ére ligne de parité	(*b*) 2éme ligne de parité

Figure 3-9 : Lignes de parité du X-code.

De la Figure 3-9, nous remarquons que les indices des symboles d'une colonne à une autre et d'une ligne à une autre suivent une relation d'ordre bien déterminée. En effet, la relation peut être définie comme la fonction *successeur* sur le groupe d'entiers {0, 1, 2, … , *m*-2, *m*-1}, et tel que le *successeur(m-1)* est l'élément 0. La fonction inverse de *successeur* est *prédécesseur*. Dans le groupe d'entiers {0, 1, 2, … , *m*-2, *m*-1}, le *prédécesseur(0)* est l'élément *m*-1.

Chaque diagonale de la première (resp. deuxième) ligne de parité implique *m* symboles, dont *m*-2 symboles de données, un symbole de parité, et un symbole fictif. Ce dernier appartient à la deuxième (resp. première) ligne de parité.

Les *m* diagonales, permettant de calculer les symboles de parité de la première ligne, impliquent les symboles de données, tel que:

$$Pour \;\; i \in [0,\, m-1], \quad La\ i^{éme}\ diagonale\ est\ l'ensemble\ \{a_{i,0},\, a_{Succ(i),Succ(0)},\, \underrightarrow{\ldots}_{m\ éléments}\}$$

Les *m* diagonales, permettant de calculer les symboles de parité de la deuxième ligne, impliquent les symboles de données, tel que:

$$Pour \;\; i \in [0,\, m-1], \quad La\ i^{éme}\ diagonale\ est\ l'ensemble\ \{a_{i,0},\, a_{Succ(i),Prec(0)},\, \underrightarrow{\ldots}_{m\ éléments}\}$$

Complexité de Codage

Notons qu'un symbole de parité est le XOR de *m*-3 symboles de données. Etant donné, que notre tableau comporte 2**m* symboles de parités.

3.7.2 Processus de Décodage

Rappelons que l'algorithme proposé ne permet de prévenir que contre l'échec d'une colonne ou de deux colonnes seulement.

3.7.2.1 Récupération d'une colonne

Xu et Bruck proposent un algorithme de détection et correction d'erreurs dans une colonne. Cette faculté ne nous intéresse pas, car nous nous intéressons à l'effacement. Dans ce qui suit, nous étudions le cas d'une perte des symboles de toute une colonne.

<u>Exemple</u>: Perte de la colonne 2 avec m = 5

Dans la Figure 3-10, les symboles inconnus de la colonne 2 sont soulignés.

$a_{3,0} = \underline{a_{0,2}} \oplus a_{1,3} \oplus a_{2,4}$	$a_{4,0} = a_{0,3} \oplus \underline{a_{1,2}} \oplus a_{2,1}$
$a_{3,1} = a_{0,3} \oplus a_{1,4} \oplus a_{2,0}$	$a_{4,1} = a_{0,4} \oplus a_{1,3} \oplus \underline{a_{2,2}}$
$\underline{a_{3,2}} = a_{0,4} \oplus a_{1,0} \oplus a_{2,1}$	$\underline{a_{4,2}} = a_{0,0} \oplus a_{1,4} \oplus a_{2,3}$
$a_{3,3} = a_{0,0} \oplus a_{1,1} \oplus \underline{a_{2,2}}$	$a_{4,3} = a_{0,1} \oplus a_{1,0} \oplus a_{2,4}$
$a_{3,4} = a_{0,1} \oplus \underline{a_{1,2}} \oplus a_{2,3}$	$a_{4,4} = \underline{a_{0,2}} \oplus a_{1,1} \oplus a_{2,0}$
(*a*) Equations de codage de la 1ére ligne de parité	(*b*) Equations de codage de la 2éme ligne de parité

Figure 3-10 : Equations de Codage du Schéma X-code, *m* =5, perte de la colonne 2.

Il y'a lieu de noter que (*1*) certaines équations n'impliquent aucun symbole de la colonne 2, et (*2*) en se basant sur les équations de codage d'une seule ligne de parité nous pouvons déterminer les symboles de données de la colonne en échec, pour calculer les deux symboles de parité manquants.

Nous proposons l'Algorithme 3-4 pour récupérer des symboles d'une colonne *i* en échec.

Algorithme 3-4 : Récupération d'une colonne dans *X-code*.

1. Récupération des symboles de données suivant les équations de codage de la 1ére ligne de parité.

$$Pour\ j \in [0, m-1] \setminus \{i, i-1\},\ a_{\prec i-j-2 \succ_m, i} = a_{m-2,j} \oplus \sum_{\substack{k=0 \\ k \neq \prec i-j-2 \succ_m}}^{m-3} a_{k, \prec j+k+2 \succ_m}$$

2. Récupération du symbole $a_{m-2,i}$, appartenant à la 1ére ligne de parité.

$$a_{m-2,i} = \sum_{k=0}^{m-3} a_{k, \prec i+k+2 \succ_m}$$

3. Récupération du symbole $a_{m-1,i}$, appartenant à la 2éme ligne de parité.

$$a_{m-1,i} = \sum_{k=0}^{m-3} a_{k, \prec i-k-2 \succ_m}$$

3.7.2.2 Récupération de 2 Colonnes

L'article propose un algorithme de re-calcul du contenu de deux colonnes, que nous détaillons. En cas d'échec de deux colonnes, l'Algorithme 3-5 récupère 2*(*m*-2) symboles de données et quatre symboles de parité.

Exemple : Perte de la colonne 1 et 3, avec $m = 5$

Dans la Figure 3-11, les symboles des colonnes 1 et 3 sont soulignés.

Remarquons que, certaines équations de codage ont une seule inconnue. Ceci réduit la complexité de résolution du système d'équations. Dans une première étape, il est intéressant de déterminer les symboles de données existant dans une équation à une inconnue. Ceci revient à déterminer les diagonales incluant un seul symbole de données

parmi les symboles des deux colonnes en échec. Ainsi, si nous supposons l'échec des deux colonnes i et j, la diagonale en question doit croiser soit la colonne i ou la colonne j, et non les deux à la fois.

Rappelons que chaque diagonale de la première (resp. deuxième) ligne de parité implique m symboles, dont m-2 symboles de données, un symbole de parité, et un symbole fictif. Ce dernier appartient à la deuxième (resp. première) ligne de parité.

- Dans la 1ére ligne de parité, les diagonales ont une pente égale à 1. Ainsi, Si (a,b) et (a', b') appartiennent à la même diagonale Alors $(a' - a) = \langle b' - b \rangle_m$.

  Supposons que le $x^{\text{éme}}$ symbole de données de la $i^{\text{éme}}$ colonne est le seul symbole inconnu d'une diagonale. Cette diagonale remplit les deux conditions suivantes : (*1*) La diagonale croise la $j^{\text{éme}}$ colonne à la ligne (m-1), et (*2*) La diagonale croise la première ligne de parité à la $y^{\text{éme}}$ colonne. Ainsi, les points (x, i), $(m-1, j)$ et $(m-2, y)$ appartiennent à cette diagonale. Nous déduisons que, $x = (m-1) - (j - i)$ et $y = j - 1$. Ainsi, le symbole de parité $a_{m-2, j-1}$ détermine le symbole de données $a_{(m-1)-j-i,i}$

  En permutant i et j, nous pouvons conclure que les points (x, j), $(m-1, i)$ et $(m-2, y)$ appartiennent à la même diagonale. Nous déduisons que, $x = j - i - 1$ et $y = \langle i - 1 \rangle_m$.

  Par conséquent, le symbole de parité $a_{m-2,\langle i-1 \rangle_m}$ détermine le symbole de données.

- De la même manière, nous montrons que le symbole de parité $a_{m-1,\langle j+1 \rangle_m}$ détermine le symbole de données $a_{j-i-1,i}$. Et en permutant i et j, le symbole de parité $a_{m-1,i+1}$ détermine le symbole de données $a_{(m-1)+i-j,j}$.

(a) Equations de codage de la 1ére ligne de parité	(b) Equations de codage de la 2éme ligne de parité
$a_{3,0} = a_{0,2} \oplus \underline{a_{1,3}} \oplus a_{2,4}$	$a_{4,0} = \underline{a_{0,3}} \oplus a_{1,2} \oplus \underline{a_{2,1}}$
$\underline{a_{3,1}} = a_{0,3} \oplus a_{1,4} \oplus a_{2,0}$	$\underline{a_{4,1}} = a_{0,4} \oplus \underline{a_{1,3}} \oplus a_{2,2}$
$a_{3\ 2} = a_{0,4} \oplus a_{1,0} \oplus \underline{a_{2,1}}$	$a_{4,2} = a_{0,0} \oplus a_{1,4} \oplus \underline{a_{2,3}}$
$\underline{a_{3,3}} = a_{0,0} \oplus \underline{a_{1,1}} \oplus a_{2,2}$	$\underline{a_{4,3}} = \underline{a_{0,1}} \oplus a_{1,0} \oplus a_{2,4}$
$a_{3,4} = \underline{a_{0,1}} \oplus a_{1,2} \oplus \underline{a_{2,3}}$	$a_{4,4} = a_{0,2} \oplus \underline{a_{1,1}} \oplus a_{2,0}$

Figure 3-11 : Equations de Codage du Schéma X-code, *m* =5, perte des colonnes 1 et 3.

L'algorithme proposé se déroule en trois étapes : (1) Récupération des symboles de données, qui appartiennent à une équation à une inconnue ; (2) Récupération des autres symboles de données ; (3) Récupération des quatre symboles de parité, appartenant aux colonnes i et j, étant en échec.

Algorithme 3-5 : Récupération de 2 colonnes dans *X-code.*

1ére étape

Le tableau ci-dessous énumère les symboles de parités permettant de déterminer les symboles de données, appartenant à une équation à une inconnue.

Symboles de parité	*Symbole de données*

$$\xrightarrow{déter\min e}$$

$a_{m-2,j-1}$	$a_{(m-1)-j-i,i}$
$a_{m-2,\prec i-1\succ_m}$	$a_{j-i-1,j}$
$a_{m-1,\prec j+1\succ_m}$	$a_{j-i-1,i}$
$a_{m-1,i+1}$	$a_{(m-1)+i-j,j}$

Ci-dessous, nous ré-écrivons l'équation de codage relative à chaque symbole de parité, pour en déduire, les symboles déterminant chacun des quatre symboles de données.

* $a_{m-2,j-1} = \sum_{k=0}^{m-3} a_{k,\prec j+k+1\succ_m}$ Cette Σ comprend le symbole $a_{(m-1)-(j-i),i}$.

$$\Rightarrow \quad a_{(m-1)-(j-i),i} = a_{m-2,j-1} \oplus \sum_{\substack{k=0 \\ k\neq(m-1)-(j-i)}}^{m-3} a_{k,\prec j+k+1\succ_m}$$

* $a_{m-2,\prec i-1\succ_m} = \sum_{k=0}^{m-3} a_{k,\prec i+k+1\succ_m}$ Cette Σ comprend le symbole $a_{j-i-1,j}$.

$$\Rightarrow \quad a_{j-i-1,j} = a_{m-2,\prec i-1\succ_m} \oplus \sum_{\substack{k=0 \\ k\neq j-i-1}}^{m-3} a_{k,\prec i+k+1\succ_m}$$

* $a_{m-2,\prec j+1\succ_m} = \sum_{k=0}^{m-3} a_{k,\prec \prec j+1\succ_m -k-2\succ_m}$ Cette Σ comprend le symbole $a_{j-i-1,i}$.

$$\Rightarrow \quad a_{j-i-1,i} = a_{m-2,\prec j+1\succ_m} \oplus \sum_{\substack{k=0 \\ k\neq j-i-1}}^{m-3} a_{k,\prec \prec j+1\succ_m -k-2\succ_m}$$

* $a_{m-1,i+1} = \sum_{k=0}^{m-3} a_{k,\prec i-k-1\succ_m}$ Cette Σ comprend le symbole $a_{(m-1)+i-j,i}$.

$$\Rightarrow \quad a_{(m-1)+i-j,i} = a_{m-1,i+1} \oplus \sum_{\substack{k=0 \\ k\neq(m-1)+i-j}}^{m-3} a_{k,\prec i-k-1\succ_m}$$

<u>*2éme étape*</u>

Une fois les quatre équations à une inconnue résolues, nous pouvons résoudre les équations à deux inconnues, et récupérer (2**m* – 8) symboles de données.

Mailles $\leftarrow$ { $(a_{(m-1)-(j-i),i}, m-2)$, $(a_{j-i-1,j}, m-2)$, $(a_{j-i-1,i}, m-1)$, $(a_{(m-1)+i-j,i}, m-1)$ }

Indéterminés $\leftarrow$ (2**m*) – 8

Tant que (*Indéterminés* ≠ 0)

  e $\leftarrow$ *Mailles.Top*

  Résoudre(*e*) (voir Algorithme 3-6) résout des équations dérivées et impliquant *e*

  Supprimer(*e*, *Mailles*) //supprime *e* car source puisée

Fin Tant que

<u>*3éme étape*</u>

Nous récupérons les symboles de parité appartenant aux colonnes i et j en échec selon les équations de codage.

$$a_{m-2,i} = \sum_{k=0}^{m-3} a_{k,\prec i+k+2\succ_m} \qquad a_{m-2,j} = \sum_{k=0}^{m-3} a_{k,\prec j+k+2\succ_m}$$

$$a_{m-1,i} = \sum_{k=0}^{m-3} a_{k,\prec i-k-2\succ_m} \qquad a_{m-1,j} = \sum_{k=0}^{m-3} a_{k,\prec j-k-2\succ_m}$$

Algorithme 3-6 : Résoudre sachant un symbole, décodage X-code.

Entrées : $a_{x,y}$,, l

Si ($l = m$-1) Alors

Le symbole $a_{x,y}$ participe au calcul du symbole de parité $a_{m-2,t}$,

d'où $t = \prec y - x - 2 \succ_m$.

Si $t \in \{i, j\}$ Alors FIN

Sinon

Si ($y = i$) Alors $\bar{y} = j$

$a_{m-2,t}$ implique un symbole $s = a_{?,j}$, tel que $? = <j\text{-}t\text{-}2>_m$.

Sinon si ($y = j$) Alors $\bar{y} = i$

$a_{m-2,t}$ implique un symbole $s = a_{?,i}$, tel que $? = <i\text{-}t\text{-}2>_m$.

Fin si

$$a_{\prec \bar{y}-t-2\succ_m, \bar{y}} = a_{m-2,t} \oplus \sum_{\substack{k=0 \\ k \neq \prec \bar{y}-t-2\succ_m}}^{m-3} a_{k,\prec t+k+2\succ_m},$$

Décrémenter *Indéterminés* de 1,

Ajouter((s, m-2), *Mailles*),

Fin si

Sinon si ($l = m$-2) Alors

Le symbole $a_{x,y}$ participe au calcul du symbole de parité $a_{m-1,t}$,

d'où $t = \prec y + x + 2 \succ_m$.

Si $t \in \{i, j\}$ Alors STOP

Sinon

Si ($y = i$) Alors $\bar{y} = j$

$a_{m-1,t}$ implique un symbole $s = a_{?,j}$, tel que $? = <t\text{-}j\text{-}2>_m$.

Sinon si ($y = j$) Alors $\bar{y} = i$

$a_{m-1,t}$ implique un symbole $s = a_{?,i}$, tel que $? = <t\text{-}i\text{-}2>_m$.

Fin si

$$a_{\prec t-\bar{y}-2\succ_m, \bar{y}} = a_{m-1,t} \oplus \sum_{\substack{k=0 \\ k \neq \prec t-\bar{y}-2\succ_m}}^{m-3} a_{k,\prec t-k-2\succ_m},$$

Décrémenter *Indéterminés* de 1,

Ajouter ((s, m-2), *Mailles*),

Fin si

Fin si

3.7.3 Conclusion

Le schéma de parité X-code est intéressant, il a une pénalité de mise à jour exactement égale à 2, et le fait que les données de parité soient distribuées sur l'ensemble des disques de la matrice distribue de manière égale la charge de mise à jour des données de parité. Xu et Bruck ont proposé un second code se basant sur la théorie des graphes: le B-code [X98, XBBW98, XB99-b].

3.8 Schéma *Schwabe-Sutherland*

Schwabe et Sutherland [SS96, SS02] proposent un schéma qui survit à l'échec de plus d'un disque. Le schéma proposé permet d'accélérer le processus de reconstruction. En effet, les unités de parité sont distribuées sur l'ensemble des disques, ce qui implique aussi que la charge de travail de reconstruction est supportée par plusieurs disques à la fois. Le schéma implique des disques de parité de tailles différentes. L'apport de l'approche est d'éviter le re-calcul du contenu des disques de parité en cas de mise à jour d'un symbole de l'axe S (comme dans le cas du schéma EVENODD décrit en §3.6).

Le schéma *mod 0* tolére l'échec de t disques. Il implique D disques de données: $d_1, \ldots, d_D$, et t disques de parité : $P_1, \ldots, P_t$. Tous les disques de données sont de taille R en nombre d'unités. Chaque disque de parité P_j comporte $R+(j-1)(D-1)$ unités.

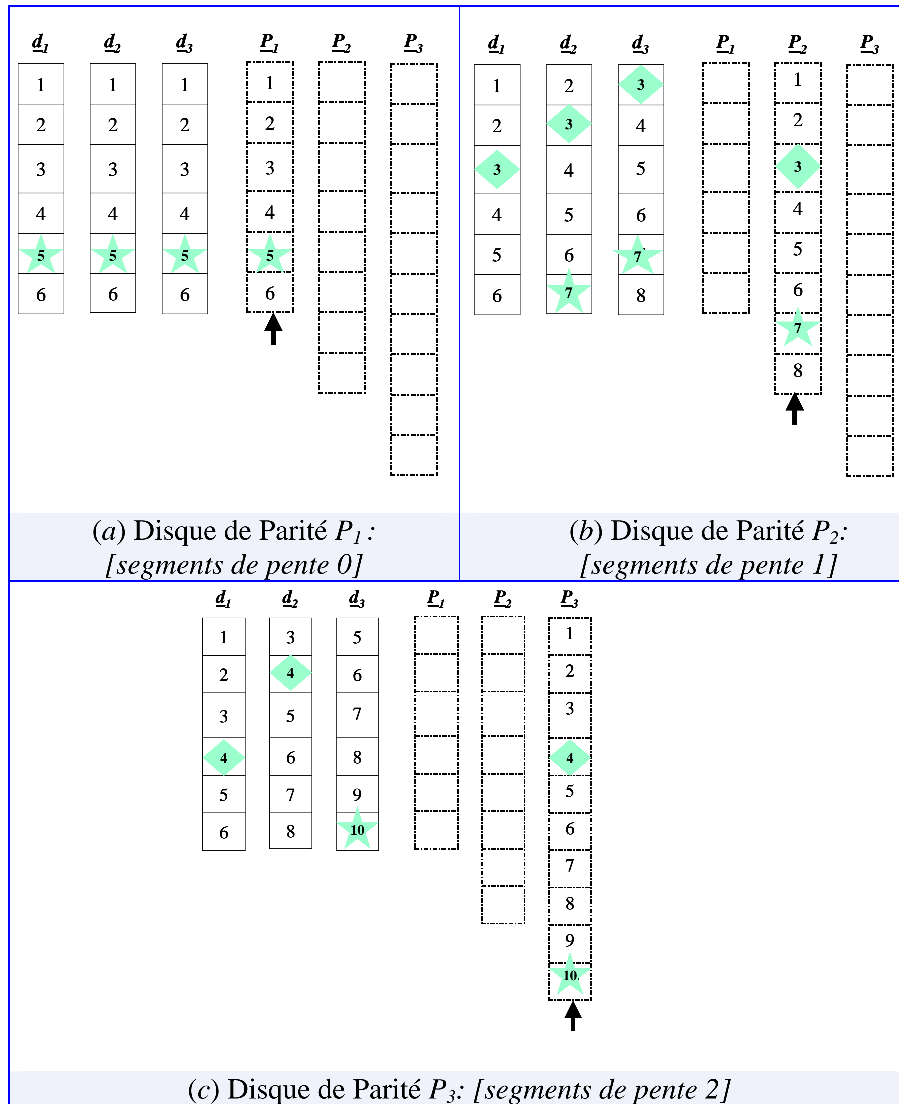
Désignons par $U_{parité}$, le nombre d'unités des t disques de parité.

$$U_{parité} = \sum_{j=1}^{t} \left(R + (j-1)(D-1) \right) = t * R + (D-1) \frac{t * (t-1)}{2}$$

Le taux de codage r est égal au rapport $U_{données} / U_{parité}$.

$$r = \frac{U_{données}}{U_{parité}} = \frac{D * R}{t * R + (D-1) \frac{t * (t-1)}{2}} = \frac{D}{t + (D-1) \frac{t * (t-1)}{2 * R}}$$

Les unités d'un disque de parité P_j sont numérotées de 1 à $R+(j-1)(D-1)$. La détermination des segments de données dépend du disque de parité à calculer. Pour un disque de parité P_i, les unités d'un disque de données sont numérotées de $1+(j-1)(i-1)$ à $R+(j-1)(i-1)$ *tel que j varie de 1 à t.*



(a) Disque de Parité P_1 : *[segments de pente 0]*

(b) Disque de Parité P_2: *[segments de pente 1]*

(c) Disque de Parité P_3: *[segments de pente 2]*

Figure 3-12: Schéma *mod 0* ; $R = 6$, $D = 3$ et $t = 3$.

Algorithme 3-7: Calcul des unités du disque de parité P_j dans le schéma de Sutherland.

Pour i: $1 \rightarrow R + (j\text{-}1)(D\text{-}1)$

  $y \leftarrow i$

  Pour x: $1 \rightarrow D$

  $p_{j\,i} \leftarrow p_{j\,i}$ XOR $d_{x\,y}$ ($d_{x\,y}$ est une unité de données ssi $y \in \{1, .., R\}$)

  $y \leftarrow y - (j\text{-}1)$

3.9 Schéma *Row-Diagonal Parity*

Corbett et al. proposent un schéma 2-disponible *RDP* (acronyme de *Row-Diagonal Parity*) [CEG+04]. Le schéma a beaucoup de ressemblances avec le schéma EVENODD [B3M95] et le schéma X-code - [XB99-a, XB99-b, X98].

Dans *RDP*, les blocs de données sont placés dans un tableau *(p-2)*(p-2)* et les blocs de parité sont respectivement sur les deux dernières colonnes. Les règles de codage, géométriquement parlant, sont suivant les diagonales de pentes 0 et –1.

En supposant l'existence d'une ligne fictive p-1, nous proposons dans l'Algorithme 3-8 les équations de codage du schéma RDP.

Algorithme 3-8: Equations de codage RDP –calcul des colonnes *p-1* et *p.*

$$\forall\, i \in [0,\ p-2],\quad \begin{cases} a_{p-1,\ i} = \sum_{t=0}^{p-1} a_{i,\, t_m} \\ a_{p,\ i} = \sum_{t=0}^{p-1} a_{t,\ <i-t>_p} \end{cases} \qquad <x>_y \;\leftrightarrow\; x \bmod y$$

Notons que, le calcul des deux disques de parité utilise le 'ou-exclusif' –XOR. A l'image du schéma EVENODD, les données du premier disque sont la parité impaire des blocs source. En ce qui concerne les données de parité du second disque de parité comme le schéma X-code, elles sont calculées selon la diagonale de pente –1, avec la particularité d'inclure les blocs de parité du premier disque de parité dans le calcul de p-2 blocs sur les p-1 blocs de parité du deuxième disque. Dans la théorie d'information, ce code est dit convolutifs. La dernière propriété contraint les mises à jour. En effet, la mise à jour d'un bloc d'un disque de données implique une mise à jour au niveau du premier disque de parité et deux mises à jour au niveau du deuxième disque de parité. En exemple, dans la Figure 3-13 la mise à jour du bloc a_{01} implique la mise à jour du bloc a_{41} du premier disque de parité, la mise à jour de a_{51} par rapport à la mise à jour de a_{01} et la mise à jour de a_{52} par rapport à la mise à jour de a_{41}. Quoi qu'il s'agisse d'une double mise à jour, la nouvelle valeur de a_{01} seule rectifiera les deux blocs pour a_{51} et a_{52}. Il y'a lieu de noter que les blocs de données d'indice 0 des disques de données et les blocs de données appartenant à la p-1 éme diagonale –(a_{04}, $a_{1\ p-1}$, …, $a_{p-2\ 1}$), n'induisent pas une double opération de mise à jour, car le bloc de parité a_{p0} et les blocs eux-mêmes ne figurent dans aucune équation de codage du second disque de parité.

Par rapport au schéma X-code, ce schéma de haute disponibilité n'est pas optimal sur le critère complexité de mise à jour.

Disques de Données				1er Disque de Parité	2éme Disque de Parité
a_{00}	a_{10}	a_{20}	a_{30}	$a_{40} = a_{00} \oplus a_{10} \oplus a_{20} \oplus a_{30}$	$a_{50} = a_{00} \oplus a_{23} \oplus a_{32} \oplus a_{43}$
a_{01}	a_{11}	a_{21}	a_{31}	$a_{41} = a_{01} \oplus a_{11} \oplus a_{21} \oplus a_{31}$	$a_{51} = a_{01} \oplus a_{10} \oplus a_{33} \oplus a_{42}$
a_{02}	a_{12}	a_{22}	a_{32}	$a_{42} = a_{02} \oplus a_{12} \oplus a_{22} \oplus a_{32}$	$a_{52} = a_{02} \oplus a_{11} \oplus a_{20} \oplus a_{41}$
a_{03}	a_{13}	a_{23}	a_{33}	$a_{43} = a_{03} \oplus a_{13} \oplus a_{23} \oplus a_{33}$	$a_{53} = a_{03} \oplus a_{12} \oplus a_{21} \oplus a_{30}$

Figure 3-13: Illustration du schéma Row-Diagonal Parity pour p = 5.

En ce qui concerne l'algorithme de décodage, Corbett et al. adoptent le même principe que l'Algorithme 3-5 de décodage proposé pour le schéma X-code, càd partant des équations de codage du deuxième disque de parité excluant les deux disques en échec. Il est à noter que quoique le schéma RDP soit apparu récemment et après les travaux de Xu et Bruck [XB99-a, XB99-b, X98], il semble que les auteurs ignorent l'existence du X-code.

3.10 Les SDDSs à Haute Disponibilité

Notons que dans les SDDSs LH* et RP* (décrites en §2.6), l'échec d'un nœud est irrémédiable. En effet, les données perdues ne peuvent pas être récupérées. Par conséquent, des structures de données distribuées et scalables à haute disponibilité ont été proposées pour répondre le critère de fiabilité. Ces structures, recensées dans la Figure 3-14, adaptent différents concepts à LH*.

- Concept de miroir : $LH*_M$ [LN96]
- Concept de fragmentation (ang. s*tripping*): $LH*_S$
- Concept de groupage : $LH*_g$, $LH*_{SA}$ & $LH*_{RS}$.

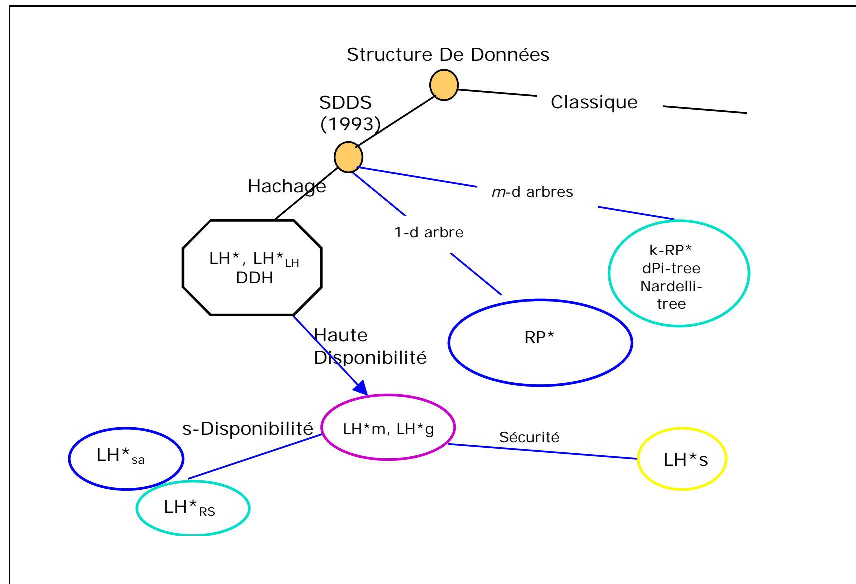


Figure 3-14: Aperçu de différentes SDDSs.

3.10.1 $LH*_M$

Un fichier distribué sur N nœuds a une probabilité de disponibilité égale à p^N, où p désigne la probabilité de disponibilité d'un nœud. Notons que, quand N croit, p^N converge vers zéro. Pour pallier cette néfaste convergence, le concept de miroirs a été appliqué à LH* [LN96]. L'existence d'un miroir implique que pour un fichier LH*, un enregistrement existe au moins dans deux cases LH*.

A chaque miroir correspond un ensemble de clients primaires. Les deux ensembles correspondant aux miroirs sont disjoints. Un client primaire *C* du miroir *F* est noté *F-client*. Un *F-client* peut accéder à *F'*, tout en ayant le statut d'un client secondaire, et vice-versa. Le fichier *F* est dit primaire, tandis que son miroir *F'* est dit secondaire. L'accès à *F'* est permis mais il est exceptionnel, il est réservé aux cas d'échec ou d'équilibre de charge entre les deux fichiers.

$LH*_M$ survit à un seul échec et est coûteuse de point de vue stockage, et ce quoique les deux fichiers soient exploités par les clients.

Dans ce qui suit, nous présentons des SDDSs à haute disponibilité, plus économiques de point de vue stockage.

3.10.2 $LH*_S$

Un fichier $LH*_S$ est constitué de k+1 fichiers segments LH*, étant S_1… S_{k+1}. Le fichier segment de parité S_{k+1} permet la récupération de données perdues, appartenant à un fichier segment de données.

Le client procède en deux étapes pour insérer un enregistrement de clé *C* et d'attributs la séquence de bits *B*: $[b_{1..}\ b_k\ b_{k+1}\ \ b_{2k}\ \dots\ b_{mk}]$. Dans une première étape, il crée k fragments ($k > 1$) tel que chaque fragment i est la concaténation des bits $[b_i\ b_{k+i}\ b_{2k+i}\ \dots]$. Chaque fragment i est envoyé par la suite au segment correspondant S_i. Dans une deuxième étape,

il génère le fragment de parité s_{k+1} qui comprend la clé C et la chaîne de bits $[b'_1\, b'_2\, b'_3 \ldots b'_m]$, tel que b_j' est égal au $\bigoplus_{j\in[1,k]} b_{ij}$, où $b_{i,j}$ désigne le *j*éme bit du fichier segment S_i.

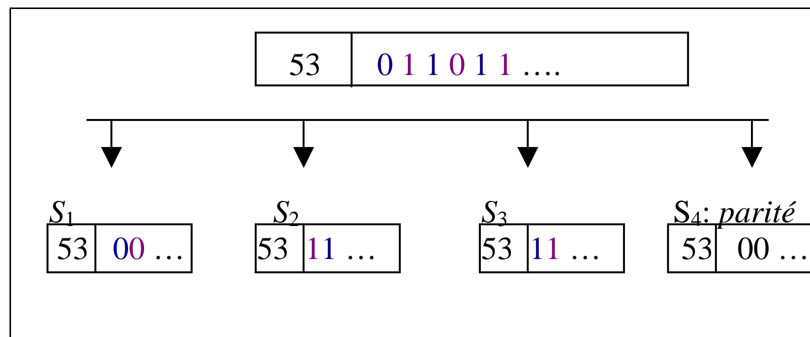


Figure 3-15: Insertion d'un enregistrement de clef 53 dans un fichier LH*S (*k* = 3).

Le schéma LH*S est 1-disponible. Il présente également un aspect de sécurité renforcé. En effet, chaque fichier segment n'a qu'un bit de k bits consécutifs de B. Désignons par x le nombre de bits manquants pour un intrus, ce dernier doit énumérer les 2^x possibilités pour reconstituer un fragment, x étant égale à $|B|(1\text{-}1/k)$.

La sécurité de LH*S est au prix d'une complexité dans la restitution et la recherche des enregistrements, puisque les manipulations client nécessitent une coopération de tous les segments. Ainsi, une variante de LH*S, se propose de faire la fragmentation au niveau attributs. Dans ce qui suit, nous décrivons des variantes de LH* surmontant l'échec de plus qu'un nœud LH*.

3.10.3 Adaptation du concept de groupage à LH*

Trois structures de données distribuées et scalables utilisant le groupage et se basant sur LH* ont été investiguées, que sont: LH**g* [LR97, L97], LH*SA[LMR98] et LH*RS [LS00, L00].

3.10.3.1 LH*g

LH**g* implante le concept de groupage. Chaque groupe est supplémenté d'un serveur de parité, calculé par XOR des serveurs de données, et permettant au groupe de survivre à l'échec d'un serveur. Un fichier LH**g* [LR97, L97] est une paire de deux fichiers LH* : F_1 et F_2: étant respectivement le fichier primaire et le fichier de parité. Une case m appartient à F1, appartient au groupe de cases $g = \lfloor m/k \rfloor$, tel que k est la taille d'un groupe de cases.
Chaque enregistrement de données occupe un rang r (ou ordre d'insertion) possède trois champs : la clef de l'enregistrement C, le couple numéro du groupe auquel et rang (g, r) et un champ *Attributs non-clé*. Quant à un enregistrement de parité, il est identifié par le couple (g, r), la liste des clés des enregistrements de rang r et appartenant au groupe g, et un champ *Bits de parité*. Ce dernier est la parité impaire des champs *Attributs non-clé* des enregistrements.

Suite à l'éclatement d'une case, un enregistrement de données sur deux migre vers la nouvelle case créée. La migration de l'enregistrement n'enclenche pas la mise à jour de l'enregistrement de parité. En effet, son identifiant ne change pas, et le champ de parité n'est pas recalculé. L'enregistrement reste donc logiquement rattaché à son groupe initial. Le contenu du fichier de parité F_2 permet de récupérer n'importe quel enregistrement de clé *C* appartenant à un groupe, et ce si et seulement si les enregistrements appartenant au même groupe logique que *C* sont disponibles.

3.10.3.2 LH*$_{SA}$

Contrairement aux schémas décrits ci-dessus, le schéma de disponibilité de LH*$_{SA}$ [LMR98] est *k*-disponible. La *k*-disponibilité du schéma LH*$_{SA}$ se matérialise par le fait qu'une case de données appartient à plusieurs groupes de parité. L'opérateur de calcul de parité –codage et décodage, est le ou-exclusif : XOR.
L'article propose des fonctions f_i tel que i varie de 1, 2, …, k et un paramètre J du fichier permettant de générer des groupes de parité. Un groupe de parité est un ensemble au maximum de k cases de données.

$$f_i : m \rightarrow g_i \quad tel\ que$$
$$g_i = (m \bmod k^{i-1}) + \left\lfloor m / k^i \right\rfloor$$
$$i \in [1..J]$$

Les groupes générés par chaque fonction sont :

$$f_1 : (m, m+1, m+2, ..., m+(k-1)) \quad pour \quad m = 0,...,k,... \quad pas = k^0$$
$$f_2 : (m, m+k, m+2k, ..., m+(k-1)k) \quad pour \quad m = 0,...,k-1, k^2,...,k^2+(k-1), 2k^2,... \quad pas = k^1$$
$$f_3 : (m, m+k^2, m+2k^2, ..., m+(k-1)k^2) \quad pour \quad m = 0,...,k^2-1, k^3...,k^3+(k^2-1), 2k^3,... \quad pas = k^2$$
…
$$f_i : (m, m+k^{i-1}, m+2k^{i-1}, ..., m+(k-1)k^{i-1}) \quad pour \quad m = 0,...,k^{i-1}-1, k^i,...k^i+(k^{i-1}-1), 2k^i,... \quad pas = k^{i-1}$$

A titre d'exemple pour $k = 4$ et $J = 4$, la case de données 0 appartient aux groupes f_1 – (0, 1, 2, 3) ; f_2 – (0, 4, 8, 12) ; f_3 – (0, 16, 32, 48) et f_4 – (0, 64, 128, 192).
Notons que les cases de données adhérent à un nouveau groupe de parité seulement quand un éclatement d'une case de données LH* crée la case de données de numéro logique une puissance de k: k, $k^2..k^i$…. Ainsi, un nouveau fichier de parité est crée, et dont les groupes sont générés par une nouvelle fonction de groupage.
Les auteurs proposent des algorithmes de récupération de *l* cases de données et d'un enregistrement de données en cas d'échec de *l* cases de données,

3.11 Conclusion

Tout au long de ce chapitre, nous avons exposé des systèmes de stockage de données et étudié le mécanisme par lequel ils assurent la fiabilité. La partie suivante s'intéresse à la SDDS à haute disponibilité LH*$_{RS}$, qui résout le problème de la tolérance aux pannes, par l'ajout de données de parité calculées par les codes Reed Solomon. Par leur propriétés, ces codes paraissaient en effet le mieux adaptes à nos besoins. Il s'agit d'une part d'un même schéma de calcul pour tout degré de

disponibilité *k*. Puis, ces codes sont systématiques ce qui permet à l'encodage de ne pas changer les valeurs de données. Ils sont également aisés pour travailler dans les corps de Galois CG(2^w). Enfin, le fait qu'il sont MDS, permet potentiellement d'optimiser le taux de remplissage.

PARTIE II : LH*$_{RS}$, UNE SDDS A HAUTE DISPONIBILITE

CHAPITRE

4 FONDEMENTS THEORIQUES DE LH*$_{RS}$

4.1 Introduction

Dans le présent chapitre, la première section expose les codes Reed Solomon, tel qu'ils sont décrits dans la littérature. La deuxième section décrit les fondements théoriques de la structure de données distribuée à haute disponibilité scalable: LH*$_{RS}$ [LS00]. Nous décrivons aussi les optimisations apportées aux processus de codage et décodage RS dans LH*$_{RS}$. La troisième section évoque des travaux de recherche qui ont utilisé les codes Reed Solomon.

4.2 Les Codes Reed Solomon

Le 21 janvier 1959, Irving Reed et Gustave Solomon ont soumis un article pour publication au « *Journal of the society for industrial and applied mathematics* ». En juin 1960, l'article intitulé «*Polynomial codes over certain Finite Fields*» a été publié [RS60]. L'article décrit une nouvelle classe de codes de corrections d'erreurs et d'effacements, qu'on connaît sous les codes Reed Solomon.

Plusieurs travaux de recherche ont utilisé les codes Reed Solomon dans différents domaines d'applications (télécommunication, support de stockage etc.) [LC83, R89, W91, SB92, BM93, WB94, BKL+95, SB95, R96, R97, P97, ABC97, IMT03, MTS04].

4.2.1 Principe

La Figure 4-1 montre un processus de codage, qui partant de m blocs de données, calcule $n - m$ blocs de parité. Le processus de décodage se matérialise par la reconstruction des m blocs de données, et ce sachant au moins m blocs de parité ou de données.

Un bloc est considéré comme une chaîne de symboles, tels que les symboles sont des éléments d'un corps de Galois. Les processus de codage et de décodage se basent sur une matrice de parité. Les sections suivantes présentent les corps de Galois et la matrice de parité.

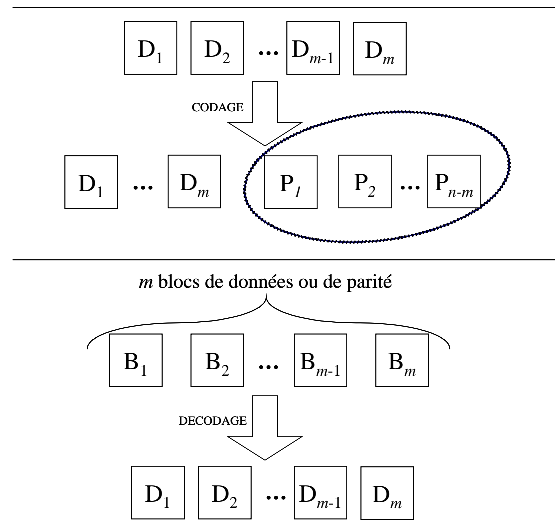


Figure 4-1: Illustration des processus de codage et décodage RS.

4.2.2 Les Corps de Galois

La théorie du codage s'appuie sur la structure algébrique de *corps fini*. Un *corps fini* a un nombre fini d'éléments, et est muni des opérations de multiplication et d'addition, et leurs inverses. Les *corps finis* sont aussi appelés *corps de Galois* (en abrégé CG), en hommage au mathématicien français *Evariste Galois* (1811-1832), qui les a découverts.

Un corps de Galois CG(q^w), de caractéristique q, contient q^w entiers, que sont 0, 1, 2, …, q^w-1, et est fermé pour l'addition et la multiplication. Ceci signifie que le résultat de l'addition, soustraction, multiplication ou division de deux éléments du corps, est un élément du corps. Ainsi, si e_1 et e_2 appartiennent à CG(q^w), alors la somme de *e1* et *e2* dans CG(q^w) est égale à la somme des deux entiers réduite modulo q^w, idem le produit de *e1* et *e2* dans CG(q^w) est égale au produit des deux entiers réduit modulo q^w. L'utilisation des corps de Galois trouve sa justification, de sa propriété de fermeture. La multiplication et la division des éléments des corps des entiers ou réels et l'arrondissement des résultats, faussent le processus de codage et décodage au risque de perdre les données source.

Le plus petit corps de Galois est le corps binaire CG(2) = {0, 1}. Les éléments d'un corps de Galois sont souvent présentés en polynôme de la forme $c_0 + c_1\mathrm{x} + c_2\mathrm{x}^2 + \ldots + c_{n-1}\mathrm{x}^{n-1}$ où n est égal à q^w et tous les c_i appartiennent à CG(2). La représentation polynomiale du *i*éme éléments de CG(q^w) est le reste de la division euclidienne de x^i par p(x). p(x) est un polynôme degré w irréductible (càd n'est pas divisible par un autre polynôme de degré inférieur) dans le corps CG(q^w).

A titre d'exemple, sachant que le polynôme $p(x) = x^3+x+1$ est irréductible dans CG(2^3), la Table 4-1 énumère l'ensemble des éléments du corps CG(2^3).

Polynôme			x^2	x^1	x^0
$x \bmod p(x)$	=	x	0	1	0
$x^2 \bmod p(x)$	=	x^2	1	0	0
$x^3 \bmod p(x)$	=	$x+1$	0	1	1
$x^4 \bmod p(x)$	=	x^2+x	1	1	0
$x^5 \bmod p(x)$	=	x^2+x+1	1	1	1
$x^6 \bmod p(x)$	=	x^2+1	1	0	1
$x^7 \bmod p(x)$	=	1	0	0	0

Table 4-1: Représentation polynomiale des éléments de CG(2^3)\{0}, *p(x)* = x^3+x+1.

L'addition de deux éléments du CG(q^w) est l'addition terme à terme. Par exemple, x^2 + (x^2+1) est égal à 1. En ce qui concerne le produit de deux polynômes nous utilisons la propriété de fermeture du corps. Par exemple, pour calculer $x*x^2$, il faut trouver à quelle classe appartient x^3, pour ce on effectue la division euclidienne de x^3 par *p(x)* qui donne x+1. La représentation sous forme de polynômes est assez limitée pour des développements théoriques.

Une deuxième méthode désigne par α une racine de *p(x)* dans un autre corps, et définit les éléments du corps de Galois en tant que puissances de α : $\{\alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{q^m-2}\}$. Les éléments dont les puissances sont de degré inférieur à w sont retenus. Le reste des éléments est obtenu en multipliant le dernier élément obtenu par α et en réduisant le résultat modulo *p(x)* si le degré est supérieur ou égal à w. L'énumération se termine en obtenant q^w éléments.

A titre d'exemple, nous énumérons les éléments de CG(2^3). Les éléments dont les puissances sont de degré inférieur à 3 sont retenus, étant {0, 1, α, α^2}.

$\alpha^3 = \alpha+1$ (car α est une racine de *p(x)* : $\alpha^3 + \alpha+1 = 0$)
$\alpha^4 = \alpha*\alpha^3 = \alpha*(\alpha+1) = \alpha^2+\alpha$
$\alpha^5 = \alpha*\alpha^4 = \alpha*(\alpha^2+\alpha) = \alpha^3+\alpha^2 = \alpha+1+\alpha^2$
$\alpha^6 = \alpha*\alpha^5 = \alpha*(\alpha^2+\alpha+1) = \alpha^3+\alpha^2+\alpha = \alpha+1+\alpha^2+\alpha = 1+\alpha^2$
$\alpha^7 = \alpha*\alpha^6 = \alpha*(\alpha^2+1) = \alpha^3+\alpha = \alpha+1+\alpha = 1$

Remarquons que $\alpha^7 = 1$. Un polynôme *p*(*x*) est primitif pour un corps de caractéristique q si $\alpha^{q^w-1} = 1$, autrement dit il n'existe pas de puissance $j < q^w$-1, telle que $\alpha^j = 1$.

Puissance de α	*Représentation polynomiale en α*	*Représentation chaîne binaire*	*Représentation Décimale*
--	0	0 0 0	0
α^0	1	0 0 1	1
α^1	α	0 1 0	2
α^2	α^2	1 0 0	4
α^3	α +1	0 1 1	3
α^4	$\alpha^2+\alpha$	1 1 0	6
α^5	$\alpha^2+\alpha+1$	1 1 1	7
α^6	α^2+1	1 0 1	5

Table 4-2: Représentation des éléments de CG(2^3).

La multiplication de deux éléments e_1 et e_2 d'un corps de Galois, s'effectue comme suit,

* Représenter les 2 éléments sous la forme de puissance de α: α^p et α^q.
* Le résultat de la multiplication est donc α^{p+q}.
* Pour trouver à quel élément du corps correspond α^{p+q}, $(p+q)$ est réduite modulo 2^w-1. Notons z le résultat du modulo.
* Il ne reste qu'à chercher la représentation binaire de l'élément α^z.

Pour effectuer les conversions (puissance → décimale) et (décimale → puissance) et, les opérations de multiplication et de division sont munies de deux tables, respectivement *log* et *antilog*., représentées ci-dessous pour CG(2^3).

Les opérations de multiplication et de division deviennent,

$$e_1 * e_2 = antilog_\alpha [(log_\alpha[e1) + log_\alpha [e2]) \text{ modulo } 2^w - 1]$$

$$e_1 \div e_2 = antilog_\alpha [(log_\alpha [e1] - log_\alpha [e2]) \text{ modulo } 2^w - 1]$$

L'opération modulo est coûteuse en temps CPU. Afin d'éviter d'effectuer le 'modulo $2^w - 1$', on double la taille de la table *antilog* au point qu'elle puisse déterminer la valeur de l'élément α^{2*2w-1} (égal à $\alpha^{2w-1} * \alpha^{2w-1}$) sans calculer le 'modulo $2^w - 1$'.

i	0	1	2	3	4	5	6	7
log[*i*]	--	0	1	3	2	6	4	5
antilog[*i*]	1	2	4	3	6	7	5	1

Table 4-3: Tables *log* et *antilog* pour CG(2^3).

On ne s'intéressera par la suite qu'aux corps de Galois de caractéristique q égale à 2, où l'addition et la soustraction de deux éléments se réduisent au ou-exclusif (modulo 2, XOR).

4.2.3 La Matrice Génératrice

Une des conditions que doit remplir la matrice génératrice est que toute sous-matrice carrée doit être inversible. La matrice de Vandermonde et la matrice de Cauchy satisfont cette propriété. M.O. Rabin [R89] évoque même toute combinaison de lignes linéairement indépendantes.

4.2.3.1 Matrice de Vandermonde

La matrice de Vandermonde V se présente comme suit. Notons par $v_1, v_2, \ldots, v_{n-1}$ le premier, le deuxième, …, le n-1$^{\text{éme}}$ élément du corps de Galois. La ligne i+1 est obtenue en multipliant la ligne i par les éléments $v_1, v_2, \ldots, v_{n-1}$. Nous obtenons la matrice de Vandermonde étendue, en ajoutant à V le vecteur de dimension m : (0, 0, …, 1).

$$V = \begin{bmatrix} 1 & 1 & \cdots & 1 & \cdots & 1 & 0 \\ 0 & v_1 & \cdots & v_i & \cdots & v_{n-1} & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & & \vdots \\ 0 & v_1^{m-1} & \cdots & v_i^{m-1} & \cdots & v_{n-1}^{m-1} & 1 \end{bmatrix}$$

Figure 4-2: Représentation de la matrice de Vandermonde.

La matrice G est la concaténation de deux matrices I_m et P, telle que $I_m(m*m)$ est la matrice d'identité. La matrice génératrice G s'obtient en effectuant des transformations élémentaires sur les lignes de la matrice de Vandermonde V.

4.2.3.2 Matrice de Cauchy

D'autres matrices peuvent être utilisées, telle que la matrice de Cauchy [R89, BKL+95, P00]. Toute sous-matrice carrée C d'une matrice de Cauchy est une matrice de Cauchy et est inversible. Vu la représentation en quotient des coefficients de la matrice (voir Figure 4-3), une matrice de Cauchy est construite avec deux ensembles $\{x_1, x_2, \ldots, x_m\}$, $\{y_1, y_2, \ldots, y_m\}$ d'éléments qui doivent remplir les deux conditions suivantes:

$$\xrightarrow{1} \forall\, i \in \{1,\ldots,m\}\ \forall\, j \in \{1,\ldots,n\} : x_i + y_j \neq 0.$$

$$\xrightarrow{2} \forall\, i, j \in \{1,\ldots,m\}, i \neq j : x_i \neq x_j \ \ et\ \forall\, i, j \in \{1,\ldots,n\}, i \neq j : y_i \neq y_j .$$

Blömer et al. [BKL+95] définissent les ensembles x et y, tels que, pour $i = 1,\ldots, 2^{w-1}$, x_i est l'élément dont la représentation binaire est celle de l'entier i-1, et y_i est l'élément dont la représentation binaire est celle de l'entier i-1+2^{w-1}. La cellule de la matrice de Cauchy correspondant à la ligne i et la colonne j, ayant pour valeur $\frac{1}{x_i + y_j}$, qui s'écrit aussi $\frac{1}{i + j + 2^w}$.

$$\begin{array}{c} j: 1\ldots n \\ i: 1\ldots m \end{array} \begin{bmatrix} \frac{1}{x_1 + y_1} & \frac{1}{x_1 + y_2} & \cdots & \frac{1}{x_1 + y_n} \\ \frac{1}{x_2 + y_1} & \frac{1}{x_2 + y_2} & \cdots & \frac{1}{x_2 + y_n} \\ \vdots & \vdots & \ddots & \vdots \\ & & \boxed{\frac{1}{x_i + y_j}} & \\ \frac{1}{x_{m-1} + y_1} & \frac{1}{x_{m-1} + y_2} & \cdots & \frac{1}{x_{m-1} + y_n} \\ \frac{1}{x_m + y_1} & \frac{1}{x_m + y_2} & \cdots & \frac{1}{x_m + y_n} \end{bmatrix}$$

Figure 4-3: Représentation de la matrice de Cauchy.

4.2.4 Code Systématique

Un code systématique a une matrice génératrice de la forme $[I_m|P]$ -concaténation de deux matrices I_m et P, telle que $I_m(m*m)$ est la matrice d'identité et $P(m*(n-m))$ est la matrice de parité. Ce type de matrice encode les données source de sorte que les données de parité reproduisent les données source.

Les transformations élémentaires utilisées pour écrire une matrice sous sa forme canonique sont au nombre de trois:

1. Permutation de deux lignes : L_i et L_j.
2. Une ligne L_i peut être remplacée par $c*L_i$ avec $c \neq 0$.
3. Une ligne L_i peut être remplacée par $L_i + c* L_j$ avec $j \neq i$ et $c \neq 0$.

L'Algorithme 4-1 consiste en *m* itérations. D'abord, nous devrions nous assurer que toutes les cellules de coordonnées (i,i) sont différentes de 0. Pour ce, dans le cas ou $e_{i,i}$ est égal à 0, il faudrait chercher une ligne *i'* tel que $i' > i$ et $e_{i,i'} \neq 0$, pour échanger la ligne *i* et la ligne *i'*. A chaque itération, (1) en première étape, nous cherchons à mettre à 1 la cellule $e_{i,i}$. Ceci revient à remplacer la ligne L_i par $(e_{i,i}^{-1} * L_i)$. L'élément $e_{i,i}^{-1}$ est l'inverse multiplicatif de $e_{i,i}$ (càd $e_{i,i}^{-1} * e_{i,i} = 1$) ; (2) en deuxième étape, nous cherchons à mettre à 0 les cellules $e_{k,i}$. Ceci revient à remplacer chaque ligne L_k par $L_k - (e_{k,i} * L_i)$.

Algorithme 4-1: Diagonalisation d'une Matrice G.

Pour *i* variant de 0 à *m*-1 Faire
  Si $e_{i,i} = 0$ Alors
    Chercher $i' > i$, tel que $e_{i,i'} \neq 0$
    Echanger les lignes *i* et *i'*
  Fin Si
 $L_i \leftarrow L_i * e_{i,i}^{-1}$
 Pour *k* variant de 0 à *m*-1, tel que $k \neq i$ Faire
 $L_k \leftarrow L_k - (e_{k,i} * L_i)$

Dans plusieurs références, cette étape est omise et la matrice d'identité $I_m(m \times m)$ est directement concaténée à la matrice de Vandermonde ou à la matrice de Cauchy.

4.2.5 Codage RS

La Figure 4-4 montre le processus de codage à l'échelle de symboles. Nous supposons que les blocs de données ont la même taille, autrement nous considérons les symboles nuls. Un bloc est considéré comme une chaîne de *t* symboles. Les blocs de parité sont générés par Algorithme 4-2 un symbole à la fois à partir des blocs de données.

Etant donnés *m* blocs de données et la matrice *P*, l'Algorithme 4-2 produit $m \times n$ symboles de parité. L'algorithme s'exécute comme suit, à chaque itération *i*, le vecteur a^i désigne les $i^{émes}$ symboles des *m* blocs de données. Afin de calculer les $i^{émes}$ symboles de parité des *n* blocs de parité, le vecteur a^i est multiplié par la matrice *P*. Tel que, la multiplication de a^i par la $j^{iéme}$ colonne de *P* produit le $i^{éme}$ symbole de du bloc de parité *j*.

Si nous désirons calculer le bloc de parité de niveau *l* dans la matrice *P*, le *i*éme symbole de parité du bloc *l* est obtenu en multipliant le vecteur a^i des $i^{émes}$ symboles des *m* blocs de données par la *l*éme colonne de *P*. Le calcul de tout le bloc de parité correspond à l'itération de cette multiplication par tous les symboles de données.

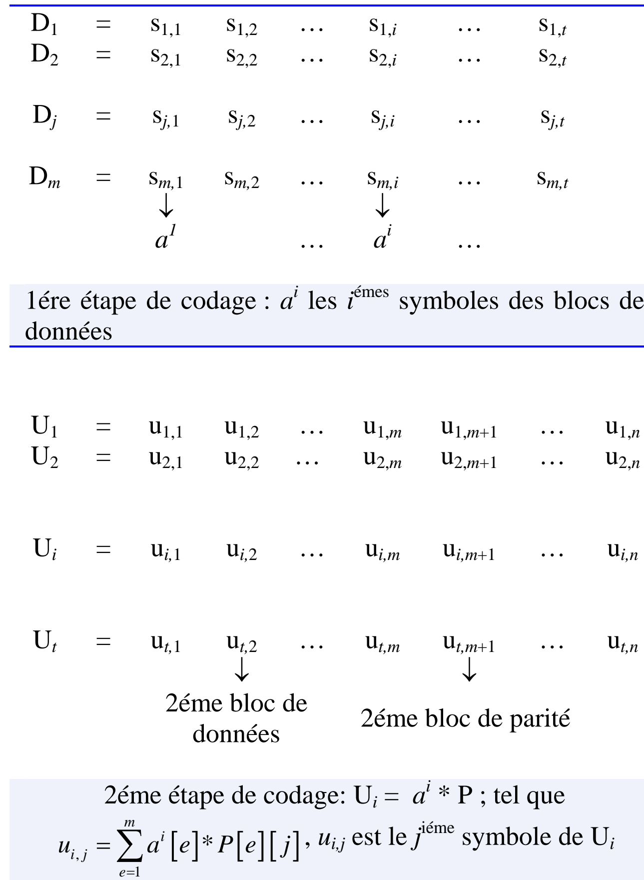


Figure 4-4: Processus de codage.

Algorithme 4-2: Codage RS.

```
Pour  i  allant de  1  à  t  Faire
   // 1ére  étape
   a^i ← (s_1i, s_2i, …, s_ji, … , s_mi) ;
   tel que s_ji désigne le i^éme symbole du j^éme bloc de données,
   // 2éme étape :  U_i = a^i *P
   Pour  j  allant de  1 à  n
       U_i [j] ← 0
       Pour  e  allant de  1 à  m
       U_i [j] += a^i[e]*P[e][j]
```

La complexité de l'Algorithme 4-2 de codage RS est en $O(t*n*m)$. La complexité de calcul d'un symbole de parité est égale à m multiplications et m-1 additions dans le corps de Galois. Le calcul de $n-m$ blocs de parité a une complexité égale à $(n-m)\times t\times m$ multiplications et $(n-m)\times t\times(m-1)$ additions dans le corps de Galois.

4.2.6 Décodage RS

Au minimum m blocs de données ou de parité doivent être disponibles pour pouvoir effectuer une récupération des f blocs de données avec $f \leq m$. L'Algorithme 4-3 montre le processus de décodage. L'algorithme commence par invertir matrice $H(m\times m)$, dont les colonnes sont extraites de P et correspondent à m blocs disponibles.

Pour ce nous utilisons le pivot de Gauss. Une fois la matrice inversée, l'étape de décodage est entamée. Cette dernière s'exécute en t itérations, et tel qu'à chaque itération i, les $i^{\text{émes}}$ symboles des blocs de données sont récupérés.

La complexité d'inversion de la matrice $H(m\times m)$ est en $O(m^3)$. Quant à la complexité de l'étape de décodage, elle est en $O(f*t*m)$. La récupération d'un symbole d'un bloc de données nécessite m multiplications et m-1 additions dans le corps de Galois. Ceci implique que la complexité de récupération de f blocs de données a une complexité égale à $f\times t\times m$ multiplications et $f\times t\times(m-1)$ additions dans le corps de Galois. Notons que, Algorithme 4-3 n'optimise pas le processus de décodage, et recalcule l'ensemble des blocs de données, nous proposons dans le cadre de notre adaptation un algorithme de décodage optimisé récupérant seulement les blocs perdus.

Le processus de décodage exige que toute sous-matrice carrée de la matrice P soit inversible, c'est une des conditions imposées sur la matrice génératrice dans la section (§4.2.3).

Algorithme 4-3: Décodage RS.

1. Collecter n'importe quels m blocs disponibles,
2. Concaténer les colonnes correspondant aux m blocs dans G, pour obtenir une matrice $H\ (m*m)$,
3. Inverser la matrice H, $H \xrightarrow{\text{Pivot de Gauss}} H^{-1}$
4. Recalculer les symboles des m blocs de données

Pour i allant de 1 à t Faire :

* $\underline{b}^i$ désigne les $i^{\text{émes}}$ symboles des m blocs de données utilisés pour la récupération,
* le processus de décodage consiste en la résolution du système linéaire suivant,

$$\underline{b}^i = H * \underline{a}^i \rightarrow \underline{a}^i = H^{-1} * \underline{b}^i$$

$$D_1[i] = a^i[1] = b^i * (H^{-1})^1 = \sum_{e=1}^{m} b^i[e] * H^{-1}[e][1]$$

$$D_2[i] = a^i[2] = b^i * (H^{-1})^2 = \sum_{e=1}^{m} b^i[e] * H^{-1}[e][2]$$

…

$$D_m[i] = a^i[m] = b^i * (H^{-1})^m = \sum_{e=1}^{m} b^i[e] * H^{-1}[e][m]$$

4.3 Le schéma de Haute disponibilité LH*$_{RS}$

Dans ce qui suit, nous décrivons la structure de données distribuée et scalable à haute disponibilité LH*$_{RS}$ proposée par Litwin et Schwarz [LS00], en particulier le fichier LH*$_{RS}$, son évolution, le calcul de parité et la récupération. Nous soulignons également la nouvelle matrice de parité, ainsi que les optimisations de codage et décodage.

4.3.1 Le Fichier LH*$_{RS}$

LH*$_{RS}$ est une structure de données distribuée et scalable et à haute disponibilité scalable. Elle distribue les enregistrements de données sur les cases (ou serveurs) de données en utilisant l'hachage linéaire (*ang. Linear Hashing, LH*). Tout ce qui concerne la distribution des données et la gestion des serveurs en surcharge est gérée par LH*. Cette dernière est détaillée dans §2.6.1.

La couche de haute disponibilité qu'ajoute LH*$_{RS}$ à la structure de données distribuée et scalable LH* est assuré par le concept de groupage et le calcul des données de parité par les codes Reed Solomon (§4.2).

A cet effet, la haute disponibilité dans LH*$_{RS}$ se matérialise par le concept de *groupes de parité*. Un fichier LH*$_{RS}$ est ainsi subdivisé en *groupes de parité*, où un *groupe de parité* est formé de *m* cases de données et de *k* cases de parité. Le groupe est dit alors *k*-disponible, car il peut survivre à l'échec d'au maximum *k* cases.

Le concept de *groupes de parité* à l'échelle des cases s'applique au niveau des enregistrements des cases, on parle alors de *segments de parité*. Notons que, chaque enregistrement de données a un rang *r* qui reflète sa position dans la case de données. Un groupe d'enregistrements, ou *segment de parité*, de rang *r* contient tous les enregistrements de données ayant le même rang dans le même groupe de parité. Les enregistrements de parité de clé *r* sont calculés à partir d'un groupe d'enregistrements de données de même rang et même groupe.

La Figure 4-5 illustre un groupe de quatre cases de données, et auquel sont rattachées deux cases de parité.

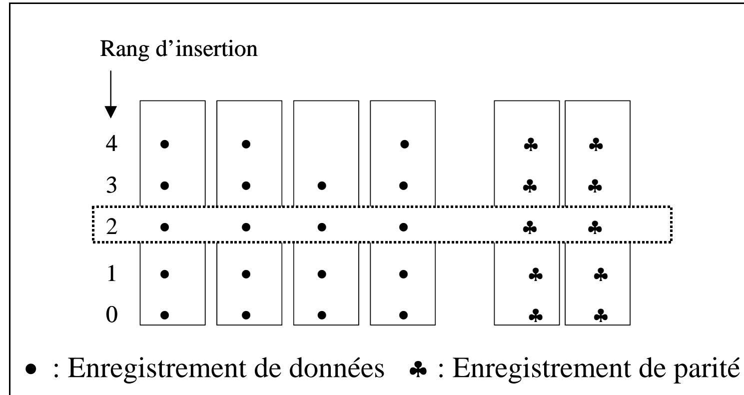


Figure 4-5: Un groupe de parité d'un fichier LH*$_{RS}$ (m = 4, k = 2).

Les structures des enregistrements doivent mettre en relation les enregistrements de données et les enregistrements de parité du même *segment de parité*. Ainsi, Litwin et Schwarz [LS00] ont proposé les structures d'enregistrements illustrées dans Figure 4-6-(*a*) et Figure 4-6-(*b*) respectivement pour un enregistrement de données et un enregistrement de parité d'un fichier LH*$_{RS}$.

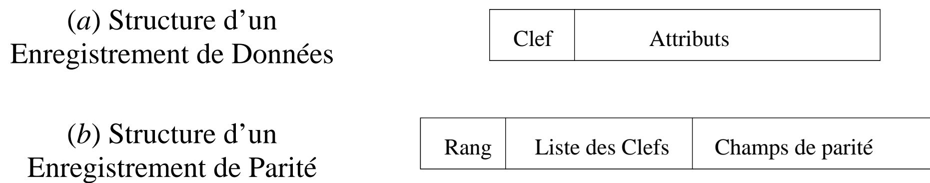


Figure 4-6: Structure des enregistrements du fichier LH*$_{RS}$.

Un enregistrement de données possède un *champ clé* et un *champ non-clé*. Quant à un enregistrement de parité, il se compose de trois champs que sont :

> **Rang,** désigne la clé de l'enregistrement de parité,
>
> **Liste des Clés,** désigne l'ensemble de *m* clés des enregistrements de données, ayant un rang *Rang* dans les cases auxquelles ils appartiennent.
>
> **Champs de parité**, ce dernier est calculé, en utilisant les codes Reed Solomon, à partir des champs *Attributs* des *m* enregistrements de données, ayant le même rang d'insertion : *Rang*, dans les cases auxquelles ils appartiennent.

4.3.2 Evolution d'un Fichier LH*$_{RS}$

La présente section s'intéresse à l'évolution de la haute disponibilité d'un fichier LH*$_{RS}$. Initialement, un fichier LH*$_{RS}$ est crée avec une case de données et une case de parité (Figure 4-7-(*a*)). La première insertion crée le premier enregistrement de parité, calculé à partir de l'enregistrement de données et des *m-1* enregistrements factices du groupe. L'enregistrement de données et l'enregistrement de parité reçoivent alors le rang 0.

Suite à d'insertions consécutives, la case de données 0 atteint sa capacité et éclate. L'éclatement transfère la moitié des enregistrements de la case de données 0 vers la case de données 1. Le fichier LH*$_{RS}$ s'étend alors sur deux cases de données (Figure 4-7-(*b*)). Notons que, suite à l'éclatement les enregistrements de données changent de rang, et ceci se répercute sur les enregistrements de parité par des suppressions et des re-insertions.

L'insertion d'enregistrements et les éclatements subséquents de cases de données, remplacent les enregistrements factices dans les cases de données par des enregistrements réels. Les éclatements créent de nouvelles cases de données 1, 2 …*m*-1. La création de la case *m* initie le second groupe de cases de données et sa première case de parité (Figure 4-7-(*c*)). Quand la taille du fichier atteint M cases de données, la probabilité d'un double échec augmente. Pour compenser le déclin de fiabilité, le nombre de cases de parité du groupe 0 et du groupe de la nouvelle case devient 2 (Figure 4-7-(*d*)). M est choisie telle qu'elle soit une puissance de 2, pour que d'une part ça coïncide avec l'éclatement de la case 0 et d'autre part, la haute disponibilité soit augmentée par le coordinateur.

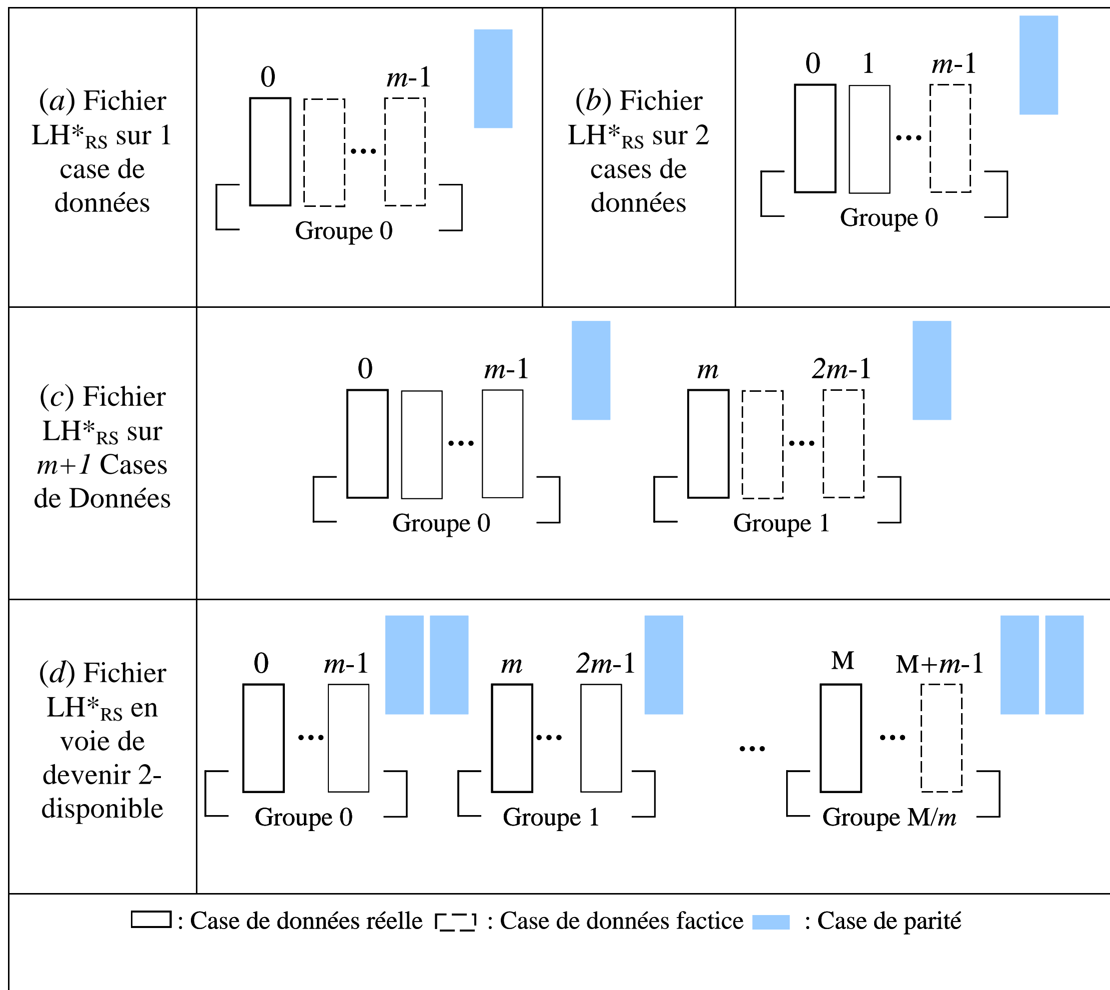


Figure 4-7: Evolution d'un fichier LH*$_{RS}$.

La haute disponibilité du fichier évolue alors à la suite de l'éclatement d'une case de données, dont le niveau de disponibilité de son groupe augmente, à moins qu'il ça été déjà effectué. Quand le fichier double de taille, càd le nombre de cases de données est égal à M*2, tous les groupes auront deux cases de parité, ainsi le fichier devient 2-disponible et tout groupe pourra pallier l'échec de deux cases.

Ce processus continue vers la *k*-disponibilité, et fait de LH*$_{RS}$ une Structure de Données Distribuée Scalable de disponibilité scalable. La valeur de M qui détermine quand le niveau de disponibilité doit augmenter est contrôlée par le coordinateur.

Litwin et Schwarz [LS00], proposent deux approches permettant de définir M, pour qu'un fichier LH*$_{RS}$ *k*-disponible devienne *k*+1- disponible, que sont :

- La stratégie de base commence l'augmentation de la *k*-disponibilité vers une *k*+1-disponibilité, quand le fichier atteint M cases. Dans la notation ci-dessus, nous aurons quand M est égal à,

  M → fichier 1-disponible,

  M^2 → fichier 2-disponible,

  ...

  M^l → fichier *l*-disponible etc.

Cette stratégie utilise des valeurs prédéfinies de M, la fiabilité est ainsi non contrôlable.

- La seconde stratégie de contrôle de fiabilité est contrôlable. En effet, A chaque fois que la case de données 0 est la prochaine case à éclater, le coordinateur évalue le niveau de fiabilité. Si ce dernier est au-dessous d'un seuil, le niveau de disponibilité du groupe 0 et du groupe auquel appartient la nouvelle case, augmente.

4.3.3 Optimisation du Codage/ Décodage RS

Tout au long de ce travail de thèse, un de nos objectifs est la réduction de la complexité du codage et du décodage RS. En comparaison, avec la matrice de parité obtenue par diagonalisation de la matrice de Vandermonde (§4.2), notre matrice de parité est plus optimisée, vu qu'elle utilise le plus de XOR. Ce qui implique un gain en performance en codage et décodage. Outre l'optimisation de la matrice de parité, nous avons optimisé le processus de codage et de décodage par le *pré-calcul du Log*.

Dans ce qui suit, nous décrivons chacune des optimisations, notamment la nouvelle matrice de parité et l'optimisation par le *pré-calcul du Log*.

4.3.3.1 Matrice de Parité

La nouvelle matrice permet la réduction de la complexité du codage et du décodage RS. Elle est caractérisée par une colonne de '1's et une ligne de '1's. Les coefficients '1's permettent de réduire une multiplication dans le corps de Galois en une opération d'addition. Rappelons-nous que la soustraction et l'addition dans un corps de Galois CG(2^w) reviennent à l'exécution d'un simple ou-exclusif (XOR).

Transformation de *P*

La colonne des '1's est obtenue en divisant chaque ligne de *P* par son premier élément, tandis que la ligne des '1's est obtenue en divisant chaque colonne de *P* par son premier élément.

A titre d'exemple, nous montrons une matrice de Vandermonde *V*, la transformation de *V* en matrice $[I_m|P]$, enfin la transformation de *P* en matrice ayant une colonne de '1's et une ligne de '1's . Le corps de Galois utilisé est CG(2^8), le nombre de lignes *m* est égal à 4.

$$\begin{bmatrix} 01 & 01 & 01 & 01 & 01 & 01 & 01 & \dots & 00 \\ 00 & 01 & 02 & 03 & 04 & 05 & 06 & & 00 \\ 00 & 01 & 04 & 05 & 10 & 11 & 14 & \ddots & 00 \\ 00 & 01 & 08 & 0f & 44 & 55 & 79 & \dots & 01 \end{bmatrix} \quad \begin{bmatrix} 01 & 00 & 00 & 00 & 1b & 1c & 12 & \dots & 7a \\ 00 & 01 & 00 & 00 & 1c & 1b & 14 & & 7a \\ 00 & 00 & 01 & 00 & 12 & 14 & 1b & \ddots & 7a \\ 00 & 00 & 00 & 01 & 14 & 12 & 1c & \dots & 7a \end{bmatrix}$$

(*a*) Matrice de Vandermonde étendue (*b*) *V* devient $[I_m|P]$

$$\begin{bmatrix} 01\ 00\ 00\ 00\ 01 & 01 & 01 & \cdots & 01 \\ 00\ 01\ 00\ 00\ 01 & d9 & 5c & & 46 \\ 00\ 00\ 01\ 00\ 01 & 5c & 46 & \iddots & 8f \\ 00\ 00\ 00\ 01\ 01 & ac & 7b & \cdots & c8 \end{bmatrix}$$

(*c*) *P* avec une ligne de'1's et une colonne de '1's

Figure 4-8: Exemple d'une matrice de Vandermonde transformée, cas CG(2^8) et $m = 4$.

La nouvelle matrice obtenue implique un gain en codage et décodage, exposés dans ce qui suit.

Gain en Codage

La première colonne de '1's correspond dans le schéma $LH*_{RS}$ à l'équation de codage de la première case de parité de chaque groupe de parité. Les coefficients '1's ont une conséquence directe sur le processus de codage, dont l'opération de base devient le ou-exclusif (XOR).

En ce qui concerne le premier coefficient de l'équation de codage de toute case de parité, il transforme les mises à jour des données de la première case de données en multiplication par '1', et donc ou-exclusif (XOR).

Gain en Décodage

En cas d'échec d'une seule case de données et de disponibilité de la première case de parité du groupe, le processus de décodage utilise l'équation de codage de la première case de parité, et exécute donc un décodage XOR afin de récupérer la case de données en échec. Notons que l'utilisation de l'équation de codage de la première case de parité implique les m-1 cases de données et la première case de parité du groupe.

La ligne de '1's ne favorise que le cas de récupération de la première case de données du groupe. En effet, dans le cas d'indisponibilité de la première case de parité, le contenu de la première case de données en échec est égal au contenu de n'importe quelle case de parité du groupe si les m-1 autres cases de données du groupe sont factices. Notons que les champs non-clé de certains enregistrements de la première case de données sont égaux en contenu aux champs de parité des enregistrements de même segment de parité si les m-1 autres enregistrements du segment en question sont factices.

4.3.3.2 Pré-calcul du Log

Le *pré-calcul du log* améliore les opérations de multiplications dans le corps de Galois en réduisant le nombre d'accès aux tables de multiplication *log* et *antilog*, et par conséquent réduit la complexité du codage et du décodage RS.

Optimisation du Codage

La $l$éme case de parité se code selon la $l$éme colonne de la matrice de parité. Le calcul de ses données de parité à l'échelle de symboles n'est autre que la multiplication d'un coefficient par un symbole de données. Le produit de deux éléments d'un corps de Galois a et b, est égal à $antilog[log[a] + log[b]]$ avec a le coefficient de P et b un

symbole de données. Le *pré-calcul du log* consiste en le calcul de *log*[*a*], et ce pour tout coefficient de de *l*éme colonne de la matrice de codage.

Ainsi, à son initialisation, une case de parité calcule le *log* de chaque élément de sa colonne de la matrice génératrice. Le produit de *a-log* et *b* devient *antilog*[*a-log* + *log*[*b*]]. Le pré-calcul du log nous fait gagner un accès à la table de logarithmes *log* par opération de multiplication.

Optimisation du Décodage

Notons par décodage RS^+ : un décodage RS optimisé. Le décodage RS^+ améliore l'algorithme de décodage RS (Algorithme 4-3), et consiste en le pré-calcul d'une part du *log* des éléments de la matrice *H* inversée et d'autre part du vecteur *b* en cas de récupération de plus qu'une case de données ($f > 1$). Après application du *pré-calcul du log* à la transposée de *H* et à *b*, le produit d'un coefficient de la transposée de *H* : $coefH^T$ par un symbole de *b* *symb-b* devient *antilog*[$coefH^T$*-log* + *symb-b-log*]. Cette optimisation réduit la complexité des opérations de multiplications dans l'algorithme de décodage.

4.3.3.3 Matrice de Parité Q

La matrice *Q* est obtenue en appliquant le pré-calcul du log (§4.3.3.2) à la matrice *P* (§4.3.3.1). Notons que chaque case de parité à sa création calcule la matrice de parité *P* et seulement le pré-calcul du log de la colonne lui correspondant dans *P*.

4.3.4 Calcul de Parité dans LH^*_{RS}

Dans un fichier LH^*_{RS}, le calcul des données de parité se fait à l'échelle des enregistrements. Dans ce qui suit, nous montrons le processus de codage adapté à LH^*_{RS}. Dans ce dernier cas, la case de parité se calcule à partir du contenu de son groupe de cases de données.

Le procédé de calcul de parité décrit en §4.2.5 n'est pas réel. En effet, les cases de parité sont mises à jour suite aux requêtes d'insertion, de suppression ou de mise à jour d'enregistrements de données. La mise à jour est l'opération générique puisque l'insertion et la suppression en sont des cas particuliers. En effet, une insertion est une mise à jour d'un enregistrement factice, et une suppression rend un enregistrement de données factice.

Dans ce qui suit, nous décrivons et illustrons par un exemple le traitement spécifique aux requêtes d'insertion, de suppression et de mise à jour d'un enregistrement de données. Les exemples de calcul de parité sont faits dans le corps de Galois: $CG(2^8)$ pour un groupe de taille *m* égale à 4, en utilisant la table de multiplication illustrée dans la Table 4-4 et la matrice *Q* décrite dans la section §4.3.3.3.

El.	*Log*	*El.*	*Log*	*El.*	*Log*	*El.*	*Log*	*El.*	*Log*	*El.*	*Log*	*El.*	*Log*	*El.*	*Log*
00	--	20	05	40	06	60	1e	80	07	a0	37	c0	1f	E0	cb
01	00	21	8a	41	bf	61	42	81	70	a1	3f	c1	2d	e1	59
02	01	22	65	42	8b	62	b6	82	c0	a2	d1	c2	43	e2	5f
03	19	23	2f	43	62	63	a3	83	f7	a3	5b	c3	d8	e3	b0
04	02	24	e1	44	66	64	c3	84	8c	a4	95	c4	b7	e4	9c
05	32	25	24	45	dd	65	48	85	80	a5	bc	c5	7b	e5	a9
06	1a	26	0f	46	30	66	7e	86	63	a6	cf	c6	a4	e6	a0
07	c6	27	21	47	fd	67	6e	87	0d	a7	cd	c7	76	e7	51
08	03	28	35	48	e2	68	6b	88	67	a8	90	c8	c4	e8	0b
09	df	29	93	49	98	69	3a	89	4a	a9	87	c9	17	e9	f5
0a	33	2a	8e	4a	25	6a	28	8a	de	aa	97	ca	49	ea	16
0b	ee	2b	da	4b	b3	6b	54	8b	ed	ab	b2	cb	ec	eb	eb
0c	1b	2c	f0	4c	10	6c	fa	8c	31	ac	dc	cc	7f	ec	7a
0d	68	2d	12	4d	91	6d	85	8d	c5	ad	fc	cd	0c	ed	75
0e	c7	2e	82	4e	22	6e	ba	8e	fe	ae	be	ce	6f	ee	2c
0f	4b	2f	45	4f	88	6f	3d	8f	18	af	61	cf	f6	ef	d7
10	04	30	1d	50	36	70	ca	90	e3	b0	f2	d0	6c	f0	4f
11	64	31	b5	51	d0	71	5e	91	a5	b1	56	d1	a1	f1	ae
12	e0	32	c2	52	94	72	9b	92	99	b2	d3	d2	3b	f2	d5
13	0e	33	7d	53	ce	73	9f	93	77	b3	ab	d3	52	f3	e9
14	34	34	6a	54	8f	74	0a	94	26	b4	14	d4	29	f4	e6
15	8d	35	27	55	96	75	15	95	b8	b5	2a	d5	9d	f5	e7
16	ef	36	f9	56	db	76	79	96	b4	b6	5d	d6	55	f6	ad
17	81	37	b9	57	bd	77	2b	97	7c	b7	9e	d7	aa	f7	e8
18	1c	38	c9	58	f1	78	4e	98	11	b8	84	d8	fb	f8	74
19	c1	39	9a	59	d2	79	d4	99	44	b9	3c	d9	60	f9	d6
1a	69	3a	09	5a	13	7a	e5	9a	92	ba	39	da	86	fa	f4
1b	f8	3b	78	5b	5c	7b	ac	9b	d9	bb	53	db	b1	fb	ea
1c	c8	3c	4d	5c	83	7c	73	9c	23	bc	47	dc	bb	fc	a8
1d	08	3d	e4	5d	38	7d	f3	9d	20	bd	6d	dd	cc	fd	50
1e	4c	3e	72	5e	46	7e	a7	9e	89	be	41	de	3e	fe	58
1f	71	3f	a6	5f	40	7f	57	9f	2e	bf	a2	df	5a	ff	af

Table 4-4: La table *log* de CG(2^8).

$$P = \begin{bmatrix} 01 & 01 & 01 & 01 \\ 01 & d9 & 5c & \cdot^{\cdot^{\cdot}} \\ 01 & 5c & 46 & \\ 01 & ac & 7b & \ldots \end{bmatrix} \xrightarrow{\text{Pr}\,\acute{e}-calcul\ du\ \log} Q = \begin{bmatrix} 00 & 00 & 00 & 00 \\ 00 & 60 & 83 & \cdot^{\cdot^{\cdot}} \\ 00 & 83 & 30 & \\ 00 & dc & ac & \ldots \end{bmatrix}$$

Figure 4-9: Matrices P (4×3) et Q (4×3), pour (m = 4, CG(2^8)).

4.3.4.1 Insertion d'un Enregistrement de Données

L'insertion d'un enregistrement *Ed* dans une case de données *d* à un rang *r*, implique la mise à jour des *k* enregistrements de parité du segment de parité *r*. Chaque enregistrement de parité appartenant à une case de parité de niveau *l*, met à jour (*i*) son *champ liste des clés* en ajoutant à l'indice '*d* modulo *m*', la clé de *Ed*, et (*ii*) son *champ de parité*, tel que chaque symbole d'indice *i* de son champ de parité additionne (XOR) à son ancienne valeur le $i^{éme}$ symbole de *Ed* multiplié par le coefficient $P[d \text{ modulo } m][l]$ de la matrice de parité. L'opération de multiplication est optimisée, car chaque case de parité pré-calcule le log de la colonne lui correspondant dans *P*.

La complexité d'une insertion d'un enregistrement de données dépend de la taille *t* de l'enregistrement de données à insérer et du niveau de disponibilité *k* du groupe de parité. Hormis, la première case de parité qui exécute *t* additions dans le corps de Galois quel que soit à quelle case appartient l'enregistrement *Ed*, toute case de parité exécute *t* additions et *t* multiplications dans le corps de Galois pour mettre à jour son champ de parité, et ce sauf si l'enregistrement provient de la première case de données du groupe. Dans ce cas particulier, elle exécute aussi *t* additions dans le corps de Galois.

<u>*Exemple*</u>

La Figure 4-10-(*a*) montre l'insertion d'un enregistrement *Ed*1 d'attributs non-clé 'Dauphine U…' –en hexadécimal '44 61 75 70 68 69 6e …', dans la première case de données d'un groupe de parité. L'insertion de *Ed*1 crée un nouveau groupe d'enregistrements de parité *Ep*1, *Ep*2 et *Ep*3. Au vu de la ligne de ligne de '0's de la matrice de parité *Q'* (la ligne des '1's de *P*), le contenu des enregistrements de parité n'est autre que celui de *Ed*1.

La Figure 4-10-(*b*) montre l'insertion d'un enregistrement *Ed*2 d'attributs non-clé 'Reed Solomon …' –en hexadécimal '52 65 65 64 20 53 6f …', dans la deuxième case de données d'un groupe de parité, et au même rang que *Ed*1. L'insertion de *Ed*2 met à jour le groupe d'enregistrements de parité *Ep*1, *Ep*2 et *Ep*3. Ainsi, chaque symbole du champ de parité de *Ep*1 est XORé au symbole de même indice du champ attributs non-clé de *Ed*2. Nous effectuons un simple calcul de parité grâce à la ligne des '0's de Q. De même, chaque symbole du champ de parité de *Ep*2 est XORé au produit du symbole de même indice du champ attributs non-clé de *Ed*2 par le coefficient 60. Ce dernier est le log du coefficient de pondération entre la deuxième case de données et la deuxième case de parité (d9). Enfin, chaque symbole du champ de parité de *Ep*3 est XORé au produit du symbole de même indice du champ attributs non-clé de *Ed*2 par le coefficient 83. Ce dernier est le log du coefficient de pondération entre la deuxième case de données et la troisième case de parité (5c).

La Figure 4-10-(*c*) montre l'insertion d'un enregistrement *Ed*3 d'attributs non-clé 'Rim Moussa …' –en hexadécimal '52 69 6d 20 4d 6f 75…', dans la troisième case de données d'un groupe de parité, et au même rang que *Ed*1 et *Ed*2. L'insertion de *Ed*3 met à jour le groupe d'enregistrements de parité *Ep*1, *Ep*2 et *Ep*3. Ainsi, chaque symbole du champ de parité de *Ep*1 est XORé au symbole de même indice du champ attributs non-clé de *Ed*3. Nous effectuons un simple calcul de parité grâce à la ligne des '1's. De même, chaque symbole du champ de parité de *Ep*2 est XORé au produit du symbole de même indice du champ attributs non-clé de *Ed*3 par le coefficient 83. Ce dernier est log du coefficient de pondération entre la troisième case de données et la deuxième case de parité (5c). Enfin, chaque symbole du champ de parité de *Ep*3 est XORé au produit du symbole de même indice du champ attributs non-clé de *Ed*3 par le coefficient 30. Ce dernier est le coefficient de pondération entre la troisième case de données et la troisième case de parité (46).

Enfin, la Figure 4-10-(*d*) montre l'insertion d'un enregistrement *Ed*4 d'attributs non-clé 'Theses Info…' –en hexadécimal '54 68 65 73 65 73 20 …', dans la quatrième case de données d'un groupe de parité, et au même rang que *Ed*1, *Ed*2 et *Ed*3. L'insertion de *Ed*4 met à jour le groupe d'enregistrements de parité *Ep*1, *Ep*2 et *Ep*3. Ainsi, chaque symbole du champ de parité de *Ep*1 est XORé au symbole de même indice du champ attributs non-clé de *Ed*4. Nous effectuons un simple calcul de parité grâce à la ligne des '1's. De même, chaque symbole du champ de parité de *Ep*2 est XORé au produit du symbole de même indice du champ attributs non-clé de *Ed*4 par le coefficient dc. Ce dernier est le log du coefficient de pondération entre la quatrième case de données et la deuxième case de parité (ac). Enfin, chaque symbole du champ de parité de *Ep*3 est XORé au produit du symbole de même indice du champ attributs non-clé de *Ed*4 par le coefficient ac. Ce dernier est le log du coefficient de pondération entre la quatrième case de données et la troisième case de parité (7b).

Ed1	*Ed2*	*Ed3*	*Ed4*	*Ep1*	*Ep2*	*Ep3*
44	0	0	0	44	44	44
61	0	0	0	61	61	61
75	0	0	0	75	75	75
70	0	0	0	70	70	70
68	0	0	0	68	68	68
69	0	0	0	69	69	69
6e	0	0	0	6e	6e	6e

(*a*) Insertion de *Ed*1 en hex. "44 61 75... "

Ed1	*Ed2*	*Ed3*	*Ed4*	*Ep1*	*Ep2*	*Ep3*
44	52	0	0	16	be	cb
61	65	0	0	04	9d	81
75	65	0	0	10	89	95
70	64	0	0	14	55	cc
68	20	0	0	48	4a	27
69	53	0	0	3a	4a	ba
6e	6f	0	0	01	bb	ec

(*b*) Insertion de *Ed*2 en hex. "52 65 65... "

Ed1	*Ed2*	*Ed3*	*Ed4*	*Ep1*	*Ep2*	*Ep3*
44	52	52	0	44	31	03
61	65	69	0	6d	ca	b5
75	65	6d	0	7d	b3	a4
70	64	20	0	34	1a	e4
68	20	4d	0	05	3f	3e
69	53	6f	0	55	c8	67
6e	6f	75	0	74	f2	c3

(*c*) Insertion de *Ed*3 en hex. "52 69 6d …"

Ed1	*Ed2*	*Ed3*	*Ed4*	*Ep1*	*Ep2*	*Ep3*
44	52	52	54	10	e1	ba
61	65	69	68	05	af	3a
75	65	6d	65	18	f9	5e
70	64	20	73	47	8d	fa
68	20	4d	65	60	75	c4
69	53	6f	73	26	5f	19
6e	6f	75	20	54	d6	18

(*d*) Insertion de *Ed*4 en hex. "54 68 65…"

Figure 4-10: Exemples d'insertions en ligne.

4.3.4.2 Suppression d'un Enregistrement de Données

La suppression d'un enregistrement *Ed* d'une case de données *d* à un rang *r*, implique la mise à jour des *k* enregistrements de parité du segment de parité *r*. Chaque enregistrement de parité appartenant à une case de parité de niveau *l*, met à jour (*i*) son *champ liste des clés* en supprimant à l'indice '*d* modulo *m*', la clé de *Ed*, et (*ii*) son *champ de parité*, tel que chaque symbole d'indice *i* de son champ de parité soustrait (XOR) à son ancienne valeur le $i^{éme}$ symbole de *Ed* multiplié par le coefficient *P*[*d* modulo *m*][*l*] de la matrice de parité. L'opération de multiplication est optimisée, car chaque case de parité pré-calcule le log de la colonne lui correspondant dans *P*.

La complexité d'une suppression d'un enregistrement de données dépend de la taille *t* de l'enregistrement de données à supprimer et du niveau de disponibilité *k* du groupe de parité. Hormis, la première case de parité qui exécute *t* additions dans le corps de Galois quel que soit à quelle case appartient l'enregistrement *Ed*, toute case de parité exécute *t* additions et *t* multiplications dans le corps de Galois pour mettre à jour son champ de parité, et ce sauf si l'enregistrement provient de la première case de données du groupe. Dans ce cas particulier, elle exécute aussi *t* additions dans le corps de Galois.

<u>*Exemple*</u>

La suppression de l'enregistrement de données *Ed4* se propage aux enregistrements de parité *Ep1*, *Ep2* et *Ep3*. Chaque enregistrement de parité doit supprimer de son contenu *Ed4*.

Au vu du coefficient 00 –(coefficient : 4éme case de données- 1ére case de parité), l'enregistrement de parité *Ep1* exécute un simple calcul XOR. Tel que, chaque symbole du champ de parité de *Ep*1 est XORé au symbole de même indice du champ attributs non-clé de *Ed4*.

Le deuxième enregistrement de parité *Ep2* met à jour son champ de parité, de sorte que chaque symbole du champ de parité de *Ep*2 est XORé au produit du symbole de même indice du champ attributs non-clé de *Ed4* par le coefficient dc –(coefficient : 4éme case de données- 2éme case de parité).

Idem, le troisième enregistrement de parité *Ep3* met à jour son champ de parité, de sorte que chaque symbole du champ de parité de *Ep*3 est XORé au produit du symbole de même indice du champ attributs non-clé de *Ed4* par le coefficient ac – (coefficient : 4éme case de données- 3éme case de parité).

Ed1	*Ed2*	*Ed3*	*Ed4*	*Ep1*	*Ep2*	*Ep3*
44	52	52	54	10	e1	Ba
61	65	69	68	05	af	3a
75	65	6d	65	18	f9	5e
70	64	20	73	47	8d	Fa
68	20	4d	65	60	75	c4
69	53	6f	73	26	5f	19
6e	6f	75	20	54	d6	18

(*a*) Segment de parité [*Ed1*, *Ed2*, *Ed3*, *Ed4*, *Ep1*, *Ep2*, *Ep3*].

Ed1	*Ed2*	*Ed3*	*Ed4*	*Ep1*	*Ep2*	*Ep3*
44	52	52	0	44	31	03
61	65	69	0	6d	ca	b5
75	65	6d	0	7d	b3	a4
70	64	20	0	34	1a	e4
68	20	4d	0	05	3f	3e
69	53	6f	0	55	c8	67
6e	6f	75	0	74	f2	c3

(*b*) Suppression de *Ed4* en hex. "54 68 65…" et MAJ de *Ep1*, *Ep2* et *Ep3*.

Figure 4-11: Exemple de suppression d'un enregistrement.

4.3.4.3 Mise à jour d'un Enregistrement de Données

La mise à jour du champ non-clé d'un enregistrement *Ed* de rang *r* et appartenant à une case de données *d*, implique la mise à jour des *k* enregistrements de parité du segment de parité *r*. Chaque enregistrement de parité appartenant à une case de parité de niveau *l*, met à jour son *champ de parité*, tel que chaque symbole d'indice *i* de son champ de parité (*i*) soustrait (par XOR) à sa valeur le $i^{\text{éme}}$ symbole de Ed_{avant} multiplié par le coefficient *P*[*d* modulo *m*][*l*] de la matrice de parité et (*ii*) additionne (par XOR) à sa valeur le $i^{\text{éme}}$ symbole de $Ed_{après}$ multiplié par le coefficient *P*[*d* modulo *m*][*l*] de la matrice de parité. Ce mode de mise à jour a un coût d'une suppression et d'une insertion donc 2*t additions et 2*t multiplications dans le corps de Galois.

Soit $o = d$ modulo m

$$Ep.champ\ de\ parité = Ep.champ\ de\ parité - Ed_{avant} * P[o][l] + Ed_{après} * P[o][l]$$
$$= Ep.champ\ de\ parité + (Ed_{après} - Ed_{avant}) * P[o][l]$$

Les opérateurs '-'et '+' désignent la soustraction et l'addition dans un corps de Galois CG(2^w).

Une optimisation consiste à calculer le △-enregistrement, étant ($Ed_{après}$ XOR Ed_{avant}), puis mettre à jour chaque symbole du champ de parité de *Ep* de sorte qu'il soit XORé au résultat de la multiplication du symbole de même indice du △-enregistrement par le coefficient *P*[*d* modulo *m*][*l*] –(coefficient : (*d* modulo *m*) iéme case de données-*l*éme case de parité). Cette optimisation réduit la complexité de mise à jour à 2*t additions et *t* multiplications dans le corps de Galois, et ce dans le cas où il ne s'agirait pas d'une mise à jour d'un enregistrement appartenant à la première case de parité et l'enregistrement de données n'appartient pas à la première case de données du groupe. En effet, la complexité de mise à jour dans ces cas est évaluée à 2*t additions dans le corps de Galois.

<u>*Exemple*</u>

La Figure 4-12-(*a*) montre la mise à jour du champ attributs non-clé de l'enregistrement de données *Ed*2, qui devient 'Distributed FMS' –correspondant à '44 69 73 74 72 69 62…' en hexadécimal. Intuitivement, l'ancienne valeur du champ attributs non-clé de l'enregistrement de données *Ed*2 doit être supprimée des enregistrements de parité *Ep*1, *Ep*2 et *Ep*3. Puis, la nouvelle valeur du champ attributs non-clé de l'enregistrement de données *Ed*2 est re-insérée. La procédure est simplifiée par le calcul du △-enregistrement étant le XOR de l'ancienne valeur et la nouvelle valeur du champ attributs non-clé de *Ed*2. Le △-enregistrement correspondant en hexadécimal à '16 0c 16 10 52 3a 0d …' est calculé par la deuxième case de données, et qui l'envoie aux cases de parité pour mettre à jour les enregistrements de parité de même groupe de parité que *Ed*2. A la réception du △-enregistrement, chaque case de parité met à jour l'enregistrement de parité de clef le rang qu'occupe *Ed*2 dans la deuxième case de données. Ainsi, chaque symbole du champ de parité de *Ep*1 est XORé au symbole de même indice du △-enregistrement. Nous effectuons un simple calcul de parité grâce à la ligne des '0's de la matrice *Q*. De même, chaque symbole du champ de parité de *Ep*2 est XORé au produit du symbole de même indice du △-enregistrement par le coefficient 60. Enfin, chaque symbole du champ de parité de *Ep*3 est XORé au produit du symbole de même indice du △-enregistrement par le coefficient 83.

Ed1	*Ed2*	*Ed3*	*Ed4*	*Ep1*	*Ep2*	*Ep3*
49	52	52	44	06	1c	c6
6e	65	69	69	09	6a	8d
20	65	6d	73	0e	04	22
74	64	20	74	57	9c	53
68	20	4d	72	32	8f	4b
65	53	6f	69	1c	45	9d
20	6f	75	62	59	ca	f3

(*a*) Mise à Jour de *Ed*2 en "44 69 73…"

Ed1	*Ed2*	*Ed3*	*Ed4*	*Ep1*	*Ep2*	*Ep3*
49	52	52	44	03	19	c3
6e	65	69	69	1a	79	9e
20	65	6d	73	09	03	25
74	64	20	74	46	8d	42
68	20	4d	72	23	9e	5a
65	53	6f	69	55	0c	d4
20	6f	75	62	74	e7	de

(*b*) Mise à Jour de *Ed*1 en hex. "41 72 72... "

Figure 4-12: Exemples de Mises à Jour en ligne.

La Figure 4-12-(*b*) montre la mise à jour du champ attributs non-clé de l'enregistrement de données *Ed*1, qui devient 'Array Codes' –correspondant à '41 72 72 61 79 20 43…' en hexadécimal. Intuitivement, l'ancienne valeur du champ attributs non-clé de l'enregistrement de données *Ed*1 doit être supprimée des enregistrements de parité *Ep*1, *Ep*2 et *Ep*3. Puis, la nouvelle valeur du champ attributs non-clé de l'enregistrement de données *Ed*1 est re-insérée. La procédure est simplifiée par le calcul du △-enregistrement étant le XOR de l'ancienne valeur et la nouvelle valeur du champ attributs non-clé *Ed*2. Le △-enregistrement correspondant à '05 13 07 11 11 49 2d…' en hexadécimal est calculé par la première case de données, et qui l'envoie aux cases de parité pour mettre à jour les enregistrements de parité de même groupe de parité que *Ed*1. A la réception du △-enregistrement, chaque case de parité met à jour l'enregistrement de parité de clef le rang qu'occupe *Ed*1 dans la première case de données. Ainsi, chaque symbole du champ de parité de *Ep*i (i = 1,2, 3) est XORé au symbole de même indice du △-enregistrement. Nous effectuons un simple calcul de parité grâce à la première ligne des '0's de la matrice *Q*.

4.3.5 Récupération dans LH*$_{RS}$

Pour pouvoir effectuer une récupération, au minimum *m* enregistrements de données ou de parité d'un segment d'enregistrements doivent être disponibles, l'algorithme de récupération procède comme décrit dans la section (§4.2.6). Dans ce qui suit, nous décrivons la procédure de récupération d'enregistrements, qui est généralisée en récupération de cases de données.

En supposant que le groupe est *k*-disponible, nous distinguons selon le nombre *f* de cases en échec et la disponibilité de la première case, deux cas : (1) Dans le premier cas *f* = 1 et la première case de parité est disponible, nous récupérons la case en échec en utilisant l'équation de codage de la première case de parité. Ce qui revient à l'exécution d'un simple décodage XOR (voir *Exemple 1*). (2) Autrement, l'algorithme de récupération procède comme décrit dans la section (§4.2.6). La récupération de cases revient à la récupération des enregistrements des cases et récupère un segment de parité à la fois (voir *Exemple 2*).

Exemple 1: Décodage XOR

Supposons l'échec de l'enregistrement de données *Ed*4, en utilisant l'équation de codage de l'enregistrement de parité *Ep*1, nous récupérons *Ed*4 par simple décodage XOR comme le montre la Figure 4-13.

Ed1	*Ed2*	*Ed3*	*Ed4*	*Ep1*	*Ep2*	*Ep3*
44	52	52	?	10	e1	ba
61	65	69	?	05	af	3a
75	65	6d	?	18	f9	5e
70	64	20	?	47	8d	fa
68	20	4d	?	60	75	c4
69	53	6f	?	26	5f	19
6e	6f	75	?	54	d6	18

(*a*) Indisponibilité de *Ed*4.

Ed4		*Ep1*		*Ep2*		*Ep3*		*Ed3*
54	=	10	⊕	e1	⊕	ba	⊕	52
68	=	05	⊕	af	⊕	3a	⊕	69
65	=	18	⊕	f9	⊕	5e	⊕	6d
73	=	47	⊕	8d	⊕	fa	⊕	20
65	=	60	⊕	75	⊕	c4	⊕	4d
73	=	26	⊕	5f	⊕	19	⊕	6f
20	=	54	⊕	d6	⊕	18	⊕	75

(*b*) Récupération de *Ed*4.

Figure 4-13 : Exemple de récupération d'un enregistrement par décodage XOR.

Exemple 2: Décodage RS

A titre d'exemple, nous supposons que seuls *Ed*4 -le quatrième enregistrement de données et les trois premiers enregistrements de parité Ep_i (i = 1,2,3) sont disponibles. Pour la récupération des trois enregistrements de données manquants, l'algorithme de récupération procède comme suit,

- Constituer H en concaténant les colonnes de G correspondant aux enregistrements disponibles et utilisés dans la procédure de récupération.
- Inverser H,

$$H = \begin{bmatrix} 00 & 01 & 01 & 01 \\ 00 & 01 & d9 & 5c \\ 00 & 01 & 5c & 46 \\ 01 & 01 & ac & 7b \end{bmatrix} \xrightarrow{\text{Pivot de Gauss}} H^{-1} = \begin{bmatrix} 01 & 97 & 97 & 01 \\ 55 & 70 & 24 & 00 \\ 70 & e2 & 92 & 00 \\ 24 & 92 & b6 & 00 \end{bmatrix}$$

- Calculer le *log* de chaque coefficient de H^{-1},

$$\log - H^{-1} = \begin{bmatrix} 00 & 7c & 7c & 00 \\ 96 & ca & e1 & - \\ ca & 5f & 99 & - \\ e1 & 99 & 5d & - \end{bmatrix}$$

- Récupérer les symboles des enregistrements de données : le processus s'exécute en t itérations. Tel que, t désigne la taille de l'enregistrement de parité. A chaque itération i, le processus de reconstruction calcule les f symboles des enregistrements en échec, en multipliant le vecteur b des m symboles disponibles par la matrice H^{-1}, et plus exactement *log-b* par *log*-H^{-1}. La figure ci-dessous illustre la récupération des premiers symboles de données des enregistrements Ed_i (i = 1, 2, 3).

Ed1	*Ed2*	*Ed3*	*Ed4*	*Ep1*	*Ep2*	*Ep3*
?	?	?	54	10	e1	ba
?	?	?	68	05	af	3a
?	?	?	65	18	f9	5e
?	?	?	73	47	8d	fa
?	?	?	65	60	75	c4
?	?	?	73	26	5f	19
?	?	?	20	54	d6	18

(*a*) Perte des Ed_i (i =1,2,3).

b^1		$log\text{-}b^1$
(54, 10, e1, ba)	→	(8f, 04, 59, 39)

$log\text{-}b^1 \times log\text{-}H^{-1}$ =
(8f×01)⊕(04×96)⊕ (59×ca)⊕(39×e1)=*44*
(8f×7c)⊕(04×ca)⊕ (59×5f)⊕(39×99)=*52*
(8f×7c)⊕(04×e1)⊕ (59×99)⊕(39×5d)=*52*

Ed1	*Ed2*	*Ed3*	*Ed4*	*Ep1*	*Ep2*	*Ep3*
44	*52*	*52*	54	10	e1	ba
?	?	?	68	05	af	3a
?	?	?	65	18	f9	5e
?	?	?	73	47	8d	fa
?	?	?	65	60	75	c4
?	?	?	73	26	5f	19
?	?	?	20	54	d6	18

(*b*) 1ére itération: Récupération des 1ers symboles des Ed_i.

Table 4-5 : Exemple de récupération d'enregistrements par décodage RS.

4.3.6 Gestion des Rangs dans LH*$_{RS}$

Dans la conception de Bennour [B00] de LH*$_{LH}$, chaque *case-LH* (ou page LH) a un pointeur vers la position de son premier enregistrement dans le tableau de données. Les enregistrements forment une liste chaînée, tel que chaque enregistrement renseigne sur la position du son suivant dans le tableau des données. En cas de suppression, un enregistrement est supprimé logiquement, et un champ spécifique indique que l'état de la cellule. Cependant, aucune stratégie de récupération et d'insertion dans les cellules libres du tableau de données n'a été proposée.

Dans la structure de données LH*$_{RS}$, comme illustré dans la Figure 4-5, un enregistrement de parité est calculé à partir d'enregistrements de données de même rang,. Ces derniers appartiennent à des cases de données du même groupe. A l'insertion d'un enregistrement de données dans une case, il lui est attribué un *rang d'insertion*. Ce dernier indique également à quel enregistrement de parité appartient l'enregistrement de données. Inversement à l'insertion, une suppression d'un enregistrement de données libère un rang dans une case de données. Dans les premières conceptions de LH*$_{RS}$ [LS00, L00, M00], la problématique de récupération des rangs libres n'a pas été posée. Dans ce qui suit, nous décrivons notre stratégie de gestion des rangs dans LH*$_{RS}$.

4.3.6.1 Traitement des Requêtes de Suppression

Nous avons jugé inutile la sauvegarde d'un champ indiquant si la cellule dans le tableau de données est supprimée ou non, le *champ clef* lui-même dans notre conception de LH*$_{RS}$ le ferait. A cet effet, si le *champ clef* est égal à –1, l'enregistrement de données est considéré logiquement supprimé (factice), et la cellule est libre.

Nous proposons une structure de *Liste de Rangs Libres*, contenant tous les rangs (ou positions dans le tableau de données) qui ont été libérés suite à des suppressions d'enregistrements. Cette liste est vide à la création d'une case de données.

Une variable *Rang Maximal* indique la position maximale occupée dans le tableau de données. Notons que, si l'enregistrement supprimé avait pour rang *Rang Maximal* dans la case, cette variable est décrémentée de 1. Nous avons prévu une procédure re-évaluant la valeur de *Rang Maximal*. La procédure est utile dans le cas ou les rangs précédents le *Rang Maximal*, sont supprimés avant *Rang Maximal*.

4.3.6.2 Traitement des Requêtes d'Insertion

Le traitement d'une requête d'insertion d'un enregistrement de données dépend si la structure *Liste de Rangs Libres* est vide ou non. Si la liste n'est pas vide, l'insertion se fait à un rang libre dépilé de la liste, sinon la variable *Rang Maximal* est incrémentée de 1 et l'enregistrement est inséré au rang '*Rang Maximal*'.

4.3.6.3 Création d'une Case de Parité

Afin qu'une case de parité s'auto-crée, une case de données communique à une case de parité son contenu. A la réception d'un enregistrement de clef égale –1, ce dernier est considéré factice.

4.3.6.4 Récupération d'une Case de Données

Etant donné que le gestionnaire de récupération est une case de parité, celle-ci ne récupère pas les enregistrements factices (clef égale à -1). Par contre, elle notifie la case de secours (*spare*) les rangs libres implicitement, pour que cette dernière reconstitue sa *Liste de Rangs Libres*.

4.4 Schémas de Haute Disponibilité à base des Codes Reed Solomon

Plusieurs chercheurs se sont intéressés aux codes Reed Solomon et l'ont appliqué pour prévenir contre la perte de données dans différents domaines d'applications, tel que le stockage de données, l'envoi de paquets dans les réseaux lorsque la liaison retour est coûteuse. Dans ce qui suit, nous décrivons des projets de recherche énumérés dans l'ordre de leurs publications [R89, SB92, BM93, BKL+95, SB95, R96, R97, P97, ABC97, W91].

L'article de M.O. Rabin [R89] peut être une première référence, mettant l'accent sur les différentes matrices de parité pouvant être utilisés.

P. E White de ECC Technologies [W91], décrit dans un rapport la description du RAID X, un niveau RAID qui utilise les codes Reed Solomon pour prévenir contre les échecs de disques. Le calcul de parité est fait au niveau physique

L'article de J. Plank [P97] est une bonne référence pour tout informaticien désirant comprendre les codes Reed Solomon. Dans son article, Plank propose de calculer les données de parité par les codes Reed Solomon dans les tableaux de disques –RAID. Ainsi, les disques RAID pourront survivre à plus d'un échec de disque.

L. Rizzo [R96, R97] a appliqué les codes Reed Solomon pour prévenir contre la perte de paquets dans le réseau.

Alvarez et al. [ABC97] ont proposé un schéma à haute disponibilité : DATUM (ang. *Disk Arrays with opTimal storage, Uniform declustering and Multiple failure tolerance*). DATUM utilise les codes Reed Solomon pour prévenir contre les échecs de disques dans les systèmes RAID. Les auteurs proposent aussi une suite de fonctions permettant la distribution des données de parité sur les disques (ang. *parity declustering*) pour que les disques de parité ne soient pas des goulots d'étranglement par les opérations de mise à jour.

4.4.1 Travaux de Blömer et al.

L'article [BKL+95] propose un code de récupération des effacements. Les algorithmes de codage/ décodage se basent sur les codes Reed Solomon, utilisant les matrices de Cauchy (§4.2.3.2). L'originalité de l'article réside dans le fait qu'à chaque élément du corps de Galois CF[2^L], correspond une matrice binaire (*w*w*). De ce fait, les opérations de multiplication sont réduites à des opérations d'ou-exclusif.

Dans ce qui suit, nous commençons par montrer comment représenter un élément d'un corps de Galois par une matrice binaire, puis nous nous inspirons des travaux de Blömer et al. Pour proposer un algorithme de codage et un algorithme de décodage. Notre analyse de l'article, est faite dans le cadre de notre problématique. Pour ce, nous adaptons les algorithmes de codage et décodage à une structure de données distribuée. Notons que les auteurs de l'article ont proposé le code pour pallier le problème de perte de paquets en réseaux.

4.4.1.1 Représentation Matricielle d'un élément d'un CG

Soit *p*(X), le polynôme irréductible de degré L sur CG(2)[X]. Le corps CG(2^L) est isomorphe à CG(2)[X]/ (*p*(X)), le corps des polynômes de CG(2)[X] modulo *p*(X). Chaque élément de CG(2^L) a une représentation polynomiale, $f(X) = \sum_{i=0}^{L-1} f_i \cdot X^i$, où f_i $\in$ CG(2^L), le polynôme a un degré maximum de L-1.

Le vecteur colonne ($f_0, f_1, \ldots, f_{L-1}$) sur CG(2^L) est le vecteur coefficients de l'élément $f(X) = \sum_{i=0}^{L-1} f_i \cdot X^i$.

$\forall f \in$ CG(2^L), soit $\tau(f)$ la matrice, dont la $i^{éme}$ ligne est le vecteur coefficients de $X^{i-1} \cdot f \bmod p(X)$.

Exemple :

Considérons le corps fini CG(2^4), dont $p(X) = 1 + x + x^4$ est le polynôme irréductible. L'exemple construit la matrice de l'élément F, dont la représentation polynomiale est X^C.

$Colonne\ 1 \rightarrow (X^0 \cdot X^C) \bmod p(X) = 1 + x + x^2 + x^3$: (1 1 1 1)

$Colonne\ 2 \rightarrow (X^1 \cdot X^C) \bmod p(X) = 1 + x^2 + x^3$: (1 0 1 1)

$Colonne\ 3 \rightarrow (X^2 \cdot X^C) \bmod p(X) = 1 + x^3$: (1 0 0 1)

$Colonne\ 4 \rightarrow (X^3 \cdot X^C) \bmod p(X) = 1$: (1 0 0 0)

Colonne 1..Colonne 4 forment la matrice (4*4) binaire correspondant à l'élément F du corps de Galois CG(2^4).

4.4.1.2 Algorithme de Codage

Nous décrivons l'algorithme de codage proposé par Blömer et al. dans le cadre des structures de données distribuées. L'algorithme de codage s'articule autour d'une

procédure de base décrite dans l'Algorithme 4-4. Cette dernière permet de multiplier un enregistrement de données par un coefficient.

Les lignes de la matrice de Cauchy déterminent les équations de calcul des enregistrements de parité, alors que les colonnes représentent les enregistrements de données. Ainsi, un enregistrement de parité Ep_i est calculé en utilisant l'équation ci-dessous, en supposant que les coefficients se trouvent sur la ligne correspondant à Ep_i.

$$Ep_i = \alpha * Ed_1 + \beta * Ed_2 + \delta * Ed_3 \ + \theta * Ed_4 + \cdots$$

Dans ce qui suit, nous montrons le calcul d'un enregistrement de parité Ep_i par rapport à un enregistrement de données Ed_j.

Soient,

ξ : est l'élément de la cellule (i, j) de la matrice de Cauchy,

Mξ (L×L) : la représentation matricielle binaire de ξ,

Chaque enregistrement de données est subdivisé en L segments, et chaque segment est constitué de *Nsegs* symboles. Notons qu'à chaque couple de valeurs de *ligEqn* et *colEqn*, le *ligEqn*^{iéme} segment de Ep_i est mis à jour par rapport au *colEqn*^{iéme} segment de Ed_j.

Algorithme 4-4: Multiplication de Ed_j par ξ

```
a ← 0
Pour ligEqn:  0 → L-1
  b ← 0
  Pour colEqn:  0 → L-1
     Si (Mξ [ligEqn][colEqn] = 1) Alors
        Pour seg:  0 → Nsegs-1
           Epi [ligEqn + a + seg] ^= Edj [b + seg]
    Fin Si
   b += Nsegs
a++
```

Pour améliorer les performances de codage, la condition de mise à jour est exprimée autrement. En effet, l'évaluation de l'expression (Mξ [*ligEqn*][*colEqn*] = 1) est équivalente à l'évaluation de l'expression *antilog*[*log*(ξ) + *colEqn*] & BIT[*ligEqn*][3].

Le tableau BIT contient L puissances de 2, étant 2^0, 2^1, …,2^{L-1}. La particularité de ces chiffres est que leurs expressions binaires indique une position du digit 1 dans la chaîne binaire, et ce à la position *i* en comptant à partir de la droite. *i* étant l'indice du chiffre dans le table au BIT. Par exemple, BIT[0] = 1 $\langle 0001 \rangle_2$, BIT[1] = 2 $\langle 0010 \rangle_2$, BIT[2] = 4 $\langle 0100 \rangle_2$…

[3] & : (en ang. bitwise-and operator), Chaque fois que deux expressions ont un digit 1 en commun à la même position, le résultat de l'opérateur & est à 1 à cette position. Par exemple, 0101 & 0100 = 0100.

L'évaluation de l'expression "*antilog*[*log*(ξ) + *colEqn*] & BIT[*ligEqn*]" permet de tester la valeur (0 ou 1) du $i^{éme}$ digit de l'expression binaire de "*antilog*[*log*(ξ) + *colEqn*]".

Puisque, chaque entrée de BIT ne possède qu'un 1, le résultat est non nul si l'expression *antilog*[*log*(ξ)+*colEqn*] possède un 1 à la position à laquelle une entrée de BIT possède un 1. Un résultat non nul indique qu'il y'a bien un 1 dans la cellule (*lignEqn, colEqn*) de Mξ étant la représentation matricielle de ξ.

4.4.1.3 Algorithme de Décodage

La technique de décodage décrite dans [BLK+95] ne se base pas sur l'inversion directe de la matrice. En effet, dans une première étape, les enregistrements de parité disponibles sont mis à jour par rapport aux enregistrements de données disponibles. La deuxième étape consistera à la résolution d'un système d'équations linéaires.

Supposons que Ep_i est disponible, ainsi que les enregistrements de données Ed_1 et Ed_4. L'enregistrement de parité Ep_i est calculé en utilisant l'équation ci-dessous :

$$Ep_i = \alpha * Ed_1 + \beta * Ed_2 + \delta * Ed_3 \ + \theta * Ed_4 + \cdots$$

La première étape consiste à calculer $Ep_i + \alpha * Ed_1 + \theta * Ed_4 \beta$ qui est égal à $\beta * Ed_2 + \delta * Ed_3 + \cdots$.

4.4.1.4 Matrice de Cauchy vs. Matrice de Vandermonde

Plusieurs travaux de recherche ayant eu recours aux codes Reed Solomon ont fait le choix de l'utilisation des matrices de Cauchy. D. Rubenstein [R98] a comparé par l'expérimentation les processus de codage et décodage relatifs à l'utilisation des différentes matrices : la matrice de Vandermonde et la matrice de Cauchy, et n'a pas trouvé que l'utilisation d'une matrice de Cauchy dans le codage/ décodage RS est plus rapide que codage/ décodage RS utilisant une matrice de Vandermonde.

4.4.2 Projet Koh-i-Noor de Microsoft

Le but du projet Koh-i-Noor de Microsoft [IMT03, MTS04] est de construire un système de stockage de données, distribué et à haute disponibilité, sur une architecture matérielle de type cluster. Les données de parité sont calculées en utilisant les codes Reed Solomon.

Sachant que le calcul des données de parité se base sur une matrice génératrice, les auteurs distinguent deux cas :

1. Dans le premier cas, le nombre de disques de parité n'excède pas 3, ainsi la matrice génératrice est la concaténation de la matrice de parité et des trois premières lignes de la matrice de Vandermonde utilisant seulement les puissances des trois premiers éléments du corps de Galois. Le calcul du premier disque de parité est optimal. En effet, étant donné que la ligne correspondante est un vecteur de '1's les opérations de codage se réduisent au ou-exclusif.
2. Dans le deuxième cas, le nombre de disques de parité est supérieur à trois, la matrice génératrice est obtenue à partir de la matrice de Vandermonde étendue et diagonalisée.

4.5 Conclusion

Tout au long de ce chapitre nous avons décrit les codes Reed Solomon et les fondements théoriques de LH*$_{RS}$. Nous avons également comparé notre adaptation des codes Reed Solomon à des travaux similaires. Les chapitres suivants décrivent notre gestionnaire LH*$_{RS}$ et son architecture opérationnelle.

CHAPITRE

5 LE GESTIONNAIRE LH^*_{RS}

5.1 Introduction

Dans le présent chapitre, nous décrivons l'architecture de référence *SDDS2000*, telle que conçue par F. Bennour et A. Diéne [B00, D01] respectivement pour une case LH* et une case RP*. Cette architecture a été adoptée sans remise en cause par M. Ljunström [L00] et moi-même [M00] pour une case LH^*_{RS}. Dans le cadre de cette thèse, l'architecture *SDDS2000* a évolué pour mieux répondre aux besoins de LH^*_{RS}, tant au niveau fonctionnalités, qu'au niveau performances. Ainsi, des architectures dérivées par extension de *SDDS2000* furent proposées: *SDDS-TCP*, *SDDS-Ack*, et *SDDS-IP@*.

Dans *SDDS2000*, le protocole TCP/IP est utilisé en cas d'éclatement d'une case RP* et d'envoi de messages de taille supérieure à 64 KO [D01] ou éclatement d'une case LH*[B00]. La conception de la gestion des connexions TCP/IP ne permet pas la gestion de connexions multiples. En l'occurrence, nous proposons l'architecture *SDDS-TCP*, qui incorpore une composante permettant une gestion efficace de plusieurs connexions TCP/IP à la fois par une case.

SDDS2000 comporte aussi une stratégie de contrôle de flux, [D01]. Elle remédie à la perte de messages UDP entre clients et serveurs. A l'instar de cette stratégie, nous avons proposé une stratégie de contrôle de flux pour la communication par le protocole UDP, pour notre gestionnaire de LH^*_{RS}. Le contrôle de flux concerne dans notre cas non seulement la communication client-serveur, mais aussi le cases de données-cases de parité. L'architecture *SDDS-Ack* est alors proposée, elle incorpore une composante permettant une stratégie de contrôle de flux.

Dans [B00], les adresses IP des serveurs susceptibles d'héberger les cases LH* sont prédéfinies dans une table statique. La solution statique nous suffisait au début de nos travaux, vu que notre objectif était de valider nos scénarii et nos algorithmes de codage /décodage. Nous avons simplement étendu la table statique des adresses de [B00] afin qu'elle prenne en compte également les serveurs de cases de parité et de rechange de LH^*_{RS}. L'évolution d'un fichier LH* ou LH^*_{RS} avec une table d'adresses statique saisie implique plus l'administrateur. Ainsi, l'architecture *SDDS-IP@* a été proposée, elle généralise l'allocation des cases aux nœuds, et remplace la table d'adresses statique par une table d'adresses dynamique qui évolue avec le fichier.

Dans ce qui suit nous détaillons différentes architectures, ayant toutes pour noyau l'architecture *SDDS2000*, et telle que chacune valide les fonctionnalités de sa précédente, et en est dérivée.

Nous adoptons un modèle de conception de l'architecture des serveurs et d'exécution concurrente. Le modèle exige que les différentes fonctionnalités d'une entité soient réparties entre plusieurs *threads* qui s'exécutent en parallèle. Le livre d'Alok Sinha [S96] et [CW98] sont des références de base pour la mise en œuvre d'une architecture multitâche et d'un système réparti.

5.2 Architecture SDDS2000

Dans ce qui suit, nous décrivons l'architecture du logiciel client SDDS et d'une case dans le système *SDDS2000*.

5.2.1 Client SDDS2000

Un client *SDDS2000* assure les fonctions envoi de requêtes et réception de réponses aux requêtes. Les différentes fonctionnalités d'un client sont assurées par trois *threads* : le *Thread Application*, *Thread Ecoute UDP*, et le *Thread de Traitement de Réponses*. Dans ce qui suit, nous décrivons chacun des *threads*.

5.2.1.1 Thread Application

Le *Thread Application* affiche un menu à l'utilisateur, lui permettant d'exprimer une requête ou d'exécuter un traitement. Une liste de choix est alors proposée, permettant de formuler des manipulations élémentaires sur le fichier, telles que : insertion, suppression, mise à jour ou récupération d'un enregistrement, et bien d'autres fonctionnalités telles que récupération de cases, augmentation de la haute disponibilité d'un groupe etc.

Une fois l'identifiant du traitement saisi, le *Thread Application* exécute le traitement spécifique à la commande, formate le(s) message(s), détermine leur(s) destinataire(s), et les envoie par UDP au(x) case(s) concernées. En résumé, le *Thread Application* se charge du protocole de communication avec les cases de données. Notons que la procédure d'envoi de messages au niveau du client utilise un port envoi UDP spécifique au client.

5.2.1.2 Thread Ecoute UDP

Le processus d'écoute UDP est initié à la création d'un client. Il reste à l'affût des messages entrants. Il écoute sur le *Port Ecoute UDP Client*, empile les messages reçus dans la file des réponses reçues, et signale la présence de messages à traiter au *Thread de Traitement de Réponses* par un événement *Existe Message*.

5.2.1.3 Thread de Traitement de Réponses

Le *Thread de Traitement de Réponses* reste à l'affût d'un signal de l'événement *ExisteMessage*, pour dépiler un message et le traiter. Les messages sont des réponses aux requêtes, des messages d'ajustement d'image ou des messages d'information : changement d'adresse etc.

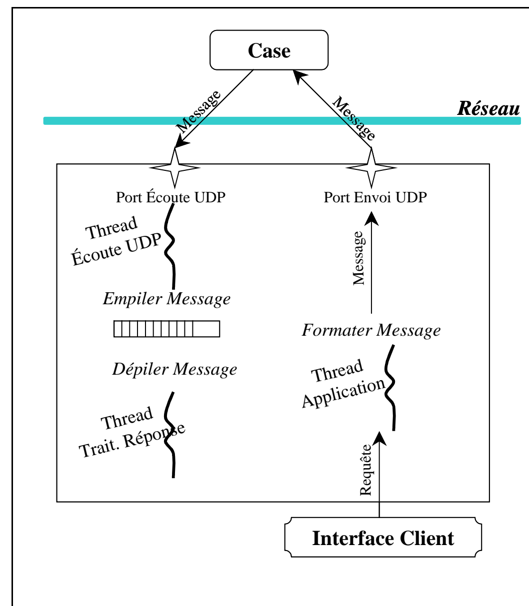


Figure 5-1: Architecture d'un Client SDDS2000.

5.2.2 Architecture d'une case SDDS2000

Une case *SDDS2000*, qu'elle soit de données ou de parité, fonctionne en mode concurrent, et se base sur la mise en œuvre du multi-tâches par l'utilisation des *threads*. Les différentes fonctionnalités d'une case sont assurées par deux types de *threads*, le *Thread Ecoute UDP* et des *Threads de Travail*, décrits dans ce qui suit,

5.2.2.1 Thread Ecoute UDP

Le *Thread Ecoute UDP* reste à l'affût des messages entrants. Il écoute sur le *Port Ecoute UDP* réservé à la case, empile les messages reçus dans la file de requêtes reçues, et signale la présence de requêtes à traiter aux *Threads de Travail* par l'événement *ExisteRequête*.

5.2.2.2 Un Pool de Threads de Travail

Le nombre de *Threads de Travail* est paramétrable. Un *Thread de Travail* reste à l'affût d'un signal de l'événement *ExisteRequête*, pour dépiler une requête, l'analyser et la rediriger ou la traiter.

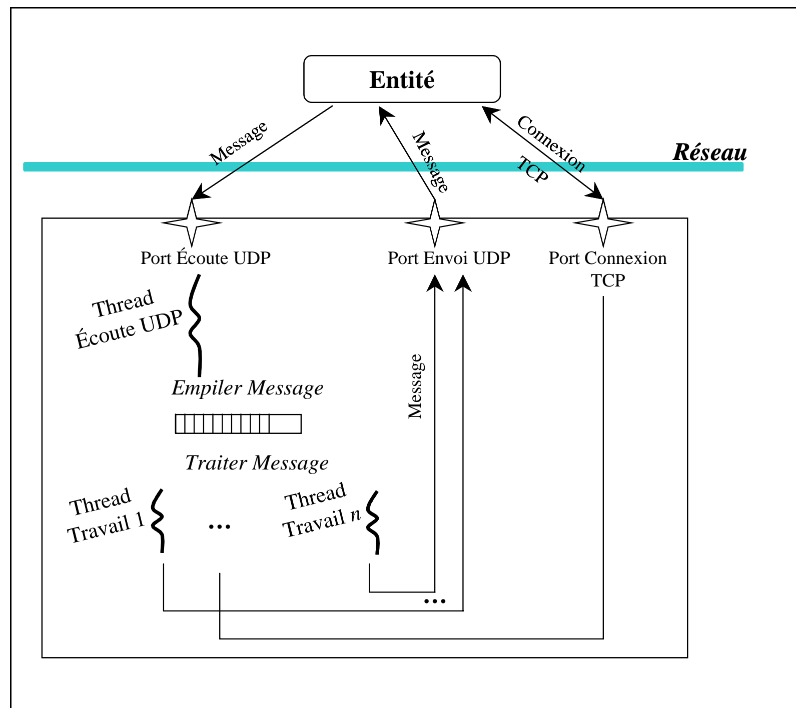
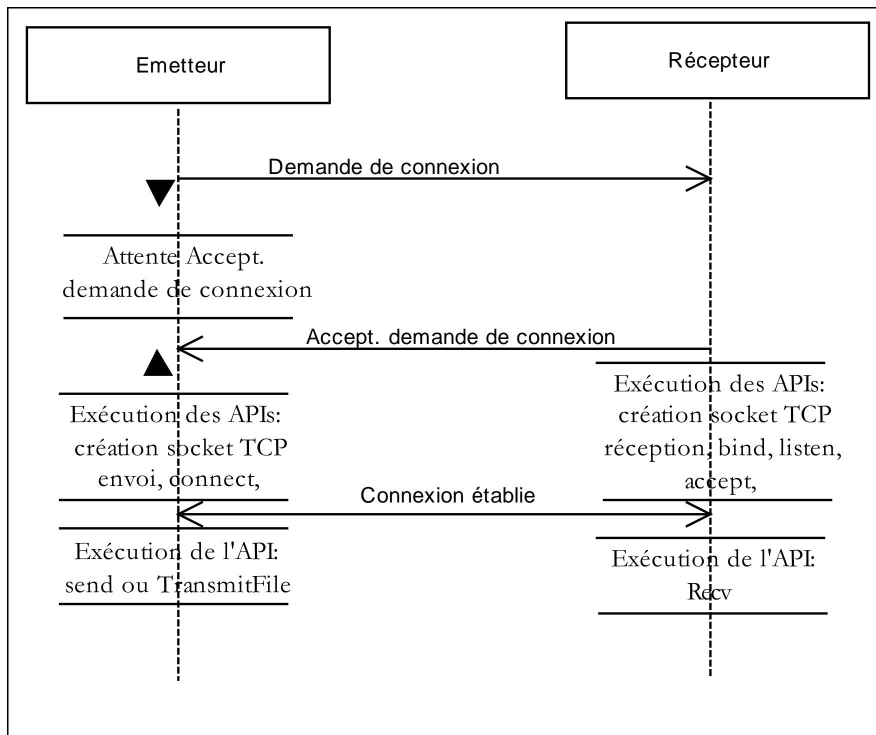


Figure 5-2: Architecture d'une case SDDS2000.

5.2.3 Les connexions TCP/IP dans SDDS2000

Chaque case possède un port de connexion TCP défini en mode programme. Afin d'établir une connexion TCP avec un pair, l'émetteur envoie par UDP le message «*Demande de connexion* », et attend la réception du message «*Connexion Acceptée* », pour préparer la connexion TCP de son coté. Il est à noter que malgré le fait que, les APIs offertes par WinSockets permettent la gestion de plusieurs connexions TCP à la fois, dans *SDDS2000*, une seule connexion TCP est acceptée à la fois.

Le Scénario 5-1 décrit l'établissement d'une connexion TCP entre deux pairs. Le scénario décrit ne tient pas compte du cas où l'un des messages «*Demande de connexion* » ou «*Connexion Acceptée* », se perd dans le réseau.



Scénario 5-1: Etablissement d'une Connexion TCP/IP dans l'architecture SDDS2000.

Notons que dans *SDDS2000* [D01-p], l'émetteur après épuisement d'un temps d'attente prédéfini re-itére sa demande de connexion en cas de perte du message «*Demande de connexion* ».

5.3 Architecture SDDS-TCP

Dans le RFC 793, [ISI81] qui présente la spécification du protocole TCP/IP, un paragraphe a suscité notre intérêt. Le paragraphe, repris ci-dessous, définit les ouvertures de 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. Dans ce cas, 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]

Cette référence nous a révélé l'idée suivante, *et si une connexion TCP était ouverte en mode passif, au niveau de chacune de nos cases ?* L'inconvénient est que la connexion inusitée consomme quand même du temps CPU. L'avantage est qu'en cas de demande de connexion TCP, nous gagnons du temps en évitant:

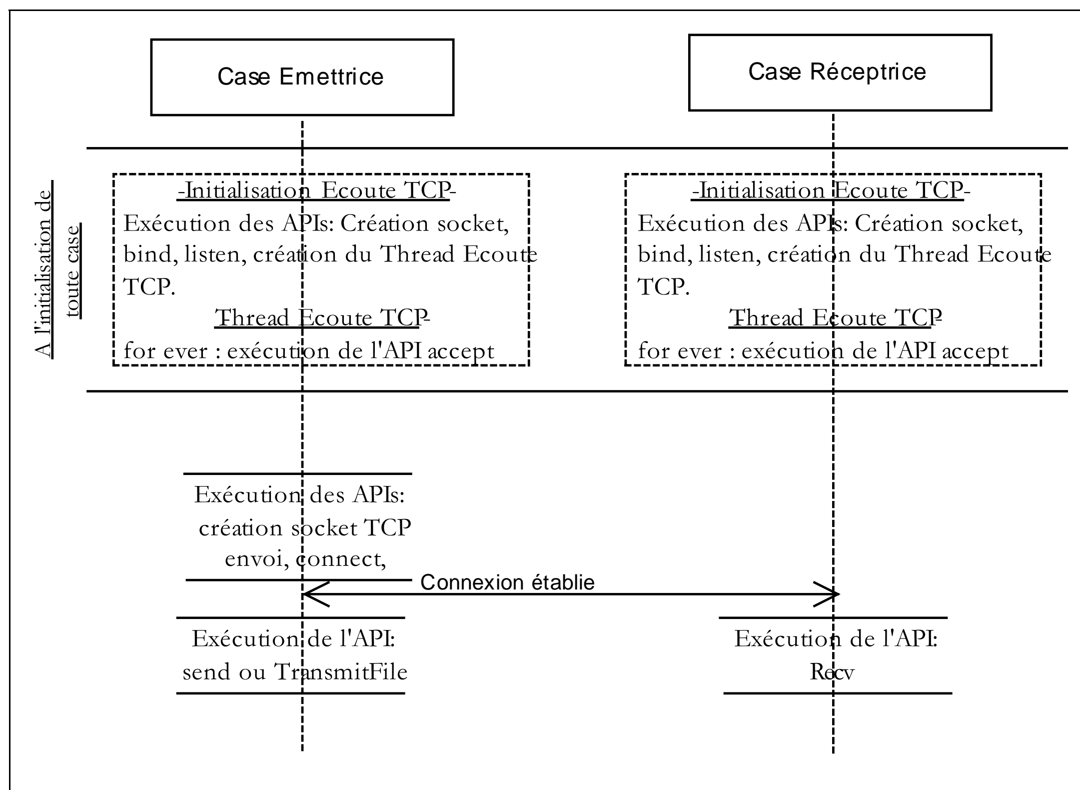
* Le temps de synchronisation des deux pairs par l'envoi du message «*Demande de connexion*», et attente du message «*Connexion Acceptée* »,
* Le traitement précédant l'établissement d'une connexion TCP, qu'effectue la case réceptrice.

Dans [MB00], les auteurs soulignent que sous les systèmes d'exploitation Windows 2000 Server et WinNT 4.0, la variable *backlog* définissant le nombre maximal de connexions TCP/IP pouvant être mises en attente (ang. *pending connections*), est égale à 200. Par contre, sous les systèmes d'exploitation Windows 2000 Professional et WinNT Workstation, *backlog* est égale à 5.

Pour une meilleure gestion des connexions TCP/IP, économisant le temps de connexion entre deux pairs, nous avons proposé une nouvelle architecture *SDDS-TCP*. La nouvelle architecture permet d'une part une ouverture passive de connexion TCP/IP, qui apprête une case à accepter une connexion TCP/IP et réceptionner un tampon, et d'autre part la gestion par l'acceptation de demandes de connexion TCP/IP concurrentes.

La nouvelle architecture ajoute le *Thread Ecoute TCP*. Ce dernier, une fois lancé, au niveau de chaque case, il reste à l'affût des demandes de connexion entrantes, les accepte, identifie le traitement à effectuer en analysant l'entête du tampon envoyé. Notons que ce *thread* écoute sur le port *Port Ecoute TCP* réservé à la case, en utilisant le protocole TCP/IP.

Le Scénario 5-2 montre l'établissement d'une connexion TCP/IP entre deux pairs dans l'architecture *SDDS-TCP*. Nous montrons également, l'initialisation d'une case qui permet une réception en mode passif des connexions TCP/IP.



Scénario 5-2: Etablissement d'une Connexion TCP dans l'architecture SDDS-TCP.

Ainsi, l'architecture d'une case devient :

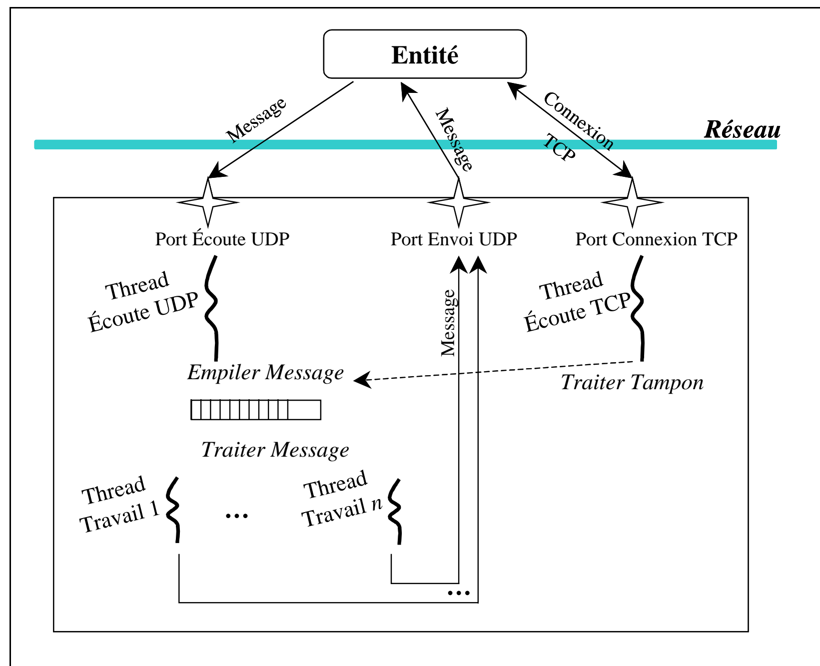


Figure 5-3: Architecture d'une case de données ou de parité dans SDDS-TCP.

5.4 Architecture SDDS-Ack

Les *WinSockets* supportent la communication avec les protocoles UDP et TCP/IP. Le protocole TCP/IP propose un service fiable basé sur la transmission de données après établissement d'une connexion entre l'émetteur et le récepteur (§2.5.2). Contrairement au protocole TCP/IP, le protocole UDP travaille en mode envoi paquets sans connexion (§2.5.1). Le protocole UDP n'est pas fiable. Nous entendons par la qualification fiable : les paquets sont reçus dans le même ordre de leur envoi, sans dédoublement, ni perte de paquets.

Les paquets sont perdus soit en transitant dans le réseau ils sont endommagés ou le réseau est saturé. Le premier cas est rare (<< 1%), donc la deuxième thèse de congestion est fort probable, c'est pour cette raison que nous avons implanté notre stratégie d'acquittement.

L'architecture baptisée *SDDS-Ack*, a été proposée dans le but d'assurer une fiabilité aux messages envoyés par le protocole UDP. Pour ce, nous proposons une stratégie de contrôle de flux à l'instar de celle proposée par A. Diène [D01]. Ainsi, nous optons pour une stratégie par fenêtre de transmission qui émule le contrôle de congestion. La fenêtre évolue en fonction des acquittements reçus. Cette stratégie permet à l'émetteur de réguler son débit d'émission.

Le principe de *Conservation de Messages* proposé par Jacobson [JK88] est appliqué dans plusieurs applications. Le récepteur ne peut pas générer d'accusès plus vite que les paquets de données transitant dans le réseau. Le protocole est *self-clocking*. Les

systèmes *self-clocking* s'ajustent automatiquement à la bande passante et à la variation des délais.

5.4.1 Stratégie d'Acquittement dans SDDS2000

Le contrôle de flux permet de se prémunir contre les pertes de messages. Le mécanisme proposé par Diéne [D01] est basé 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].

Dans *SDDS2000*, l'émetteur dispose initialement d'un crédit maximum de requêtes qu'il peut envoyer sans recevoir d'acquittements. L'épuisement de ce crédit provoque l'arrêt des envois. L'émetteur dispose d'une structure *Liste Messages Non Encore Acquittés*, de taille fixe : *Crédit Envoi*, afin de sauver les messages envoyés et en attente d'accusés. Chaque cellule de la structure sauvant les messages non encore acquittés est gérée par un *thread*, dit *Thread de contrôle de zone*. Ce dernier s'il est chargé d'un message, se met en attente de l'accusé du message. Si l'accusé est reçu avant dépassement du délai d'attente de l'accusé, le *thread* empile son identificateur dans la *File des zones libres*. Dans le cas contraire, si le nombre maximum de relances est atteint le *thread* insère son identificateur dans la *File des Relances*. La *File des Relances* est gérée par un autre *thread*, qui se charge de la re-émission des messages. En plus, des *Threads de contrôle de zone* et du *thread* de *re-émission*, un autre *thread* est requis pour l'envoi des messages, et la désignation d'un *Thread de contrôle de zone* pour la prise en charge du message.

Notre stratégie est plus simple, elle réduit le nombre de *threads*. En effet, *SDDS2000* déploie un total de (2+*Crédit Envoi*) *threads*, alors que notre stratégie d'acquittement réquisitionne seulement trois *threads*.

5.4.2 Le Protocole TRAP

Gomez, Redo et Sunderam [GRS97] proposent un protocole orienté-transaction, fiable et point à point, TRAP (acronyme de *Transaction-oriented, Reliable And Point-to-point protocol*). Le protocole TRAP est un composant clef de l'environnement de communication CLAM (acronyme de *Connectionless, Lighweight And Multiway*). TRAP est un protocole à fenêtre coulissante basé sur les paquets et non sur les octets comme TCP.

Le module de communication fiable déploie exactement trois *threads*, respectivement le *thread* de réception (ang. *Receive thread*), le *thread* d'envoi (ang. *Send thread*) et le *thread* chronométreur (ang. *Timer thread*).

- Le *thread chronométreur* planifie les retransmissions. Il s'endort pendant un laps de temps, à son réveil il vérifie la *file des événements* –une file triée par temps d'expiration.
- Le *thread d'envoi* dépile un message de la *file d'envois*, tant que cette dernière n'est pas vide. Notons que chaque message possède un identificateur permettant de prévenir contre les duplications, et assure en même temps un ordre. Chaque message est posté à la *file des évènements* et après son premier envoi à la *file des non-acquittés*. Cette dernière est une file par connexion, utile en cas de réception d'accusé, qui sera inutile de supprimer le message de la *file des évènements*.

* Le *thread de réception* réceptionne les accusés, supprime les messages correspondants de la *file des non-acquittés* et marque en tant que 'accusé reçu' afin de dé-allouer les messages correspondants par le *thread* d'envoi (en temps d'envoi) ou par le *thread* chronométreur (quand le délai a expiré) de la *file des événements* et la *file d'envois*. En effet, le *thread* de réception n'a pas accès à ces dernières files.

Les priorités des trois *threads* s'autorégulent. Le *thread chronométreur* s'exécute avec la plus grande priorité: p_t. En ce qui concerne le *thread de réception* et le *thread d'envoi*, ils s'exécutent avec une priorité: p_c; tel que $p_c < p_t$. Le *thread de réception* augmente sa priorité et celle du *thread d'envoi* à p_t dès qu'il réceptionne des messages, et les rabaisse à p_c en cas de non-réception de messages.

En ce qui concerne, la stratégie d'ordre, les messages arrivés non dans le bon ordre sont postés à la *file des hors-ordre*, et ne sont délivrées à l'application que dans le bon ordre.

Les auteurs ont mené une série d'expérimentations afin de mesurer les performances de TRAP et les comparer au PVM-TCP, PVM-UDP, LAM-TCP et LAM-UDP[4] dans des réseaux Ethernet et ATM. Les mesures démontrent la supériorité de TRAP.

5.4.3 Stratégie d'Acquittement Proposée

Comme dans *SDDS2000* [D01], l'émetteur dispose initialement d'un crédit maximum *Crédit Envoi* de requêtes qu'il peut envoyer sans recevoir d'acquittements. L'épuisement de ce crédit provoque l'arrêt des envois. L'émetteur dispose d'une structure *Liste Messages Non Encore Acquittés*, de taille fixe : *Crédit Envoi*, afin de sauver les messages envoyés et en attente d'accusés.

La réception d'un accusé constitue un événement, qui entraîne la libération d'une cellule, donc la possibilité d'envoi d'un autre message. Ainsi, à la réception d'un accusé, le message correspondant est supprimé logiquement de *Liste Messages Non Encore Acquittés*. La cellule est sue libre, dès que sa position est empilée dans la *File des Zones Libres*. L'émetteur envoie au destinataire du message, la position qu'occupe le message dans la structure *Liste Messages Non Encore Acquittés*. L'accusé informe l'émetteur sur la position qu'occupe le message dans la structure *Liste Messages Non Encore Acquittés*. Et c'est ainsi que la correspondance entre cellule de la structure *Liste Messages Non Encore Acquittés* et l'accusé de réception reçu, est faite. Cette stratégie d'échange est conçue afin d'éviter un parcours séquentiel de la structure *Liste Messages Non Encore Acquittés*, à la recherche du message correspondant à l'accusé reçu.

Les zones libres, étant les indices des cellules libres, de la structure *Liste Messages Non Encore Acquittés* sont sauvées dans une liste chaînée *Zones Libres*. Cette dernière fonctionne en mode FIFO (ang. *First In First Out*).

---

[4] LAM-MPI -acronyme de *Message Passing Interface* et PVM -acronyme de *Parallel Virtual Machine* : ce sont des bibliothèques utilisées pour exécuter des programmes parallèles sur la grappe dans le modèle de programmation par passage de messages. Pour plus d'info. Référer à http://www.lcp.u-psud.fr/Pageperso/teuler/mpi.pdf, http://www.lam-mpi.org/

A chaque message est associé un compteur de temps : *Temps Envoi*, qui est initialisé à l'envoi du message. Cela permet à l'émetteur de relancer les messages, pour lesquels les acquittements n'ont pas été reçus, au bout d'un temps prédéfini *Délai Accusé* qui a écoulé après le dernier envoi. Un nombre maximum de relances : *Nombre Max Relances* est fixé pour éviter les relances inutiles, notamment en cas de panne de serveur ou de rupture d'un lien de communication.

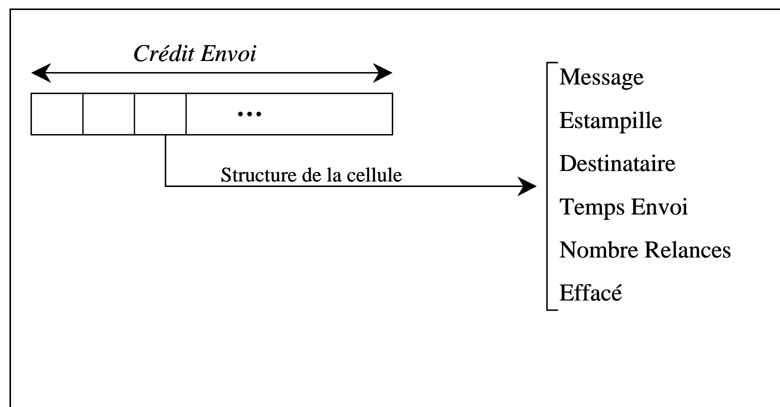


Figure 5-4: Structure de *Liste Messages Non Encore Acquittés*.

Notre protocole de communication fiable se basant sur UDP, est défini entre (*1*) clients et cases de données, et (*2*) entre cases de données et cases de parité. Actuellement, il traite les messages d'insertion entre clients et cases de données, et les messages de mise à jour entre cases de données et cases de parité. Mais, peut être généralisé pour englober tout message nécessitant un acquittement.

Dans ce qui suit, nous décrivons l'implantation de ce mécanisme au niveau du client et d'une case de données.

5.4.4 CFAM au Niveau du Client

Le mécanisme de gestion des acquittements au niveau du client est assuré par des interactions entre les threads suivants : le *Thread Application*, le *Thread de Traitement de Réponses* et le *Thread de Ré-émission de Messages*.

5.4.4.1 Thread Application

Le *Thread Application* est chargé de l'envoi de messages. Il reste à l'affût de l'état signalé de l'événement *ExisteZoneLibre*, qui indique qu'il existe au moins une cellule *c* libre d'indice *i* dans *Liste Messages Non Encore Acquittés*. Le *thread* formate le message, détermine son destinataire, et l'ajoute à la liste *Liste Messages Non Encore Acquittés*. Et ce, en initialisant les différents champs {*Message*, *Estampille*, *Destinataire*, *Temps Envoi*, *Nombre Relances*, *Libre*} resp. aux valeurs {contenu message, estampille, destinataire, 🕒, 0, '*n*'}. Dès que le message envoyé, le crédit d'envoi *Fenêtre* 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.

Notons que le modèle d'estampille utilisé est un numéro requête, qui est ré-initialisé à 1 dès qu'il atteint une valeur maximale (suffisante en fonction du débit client).

5.4.4.2 Thread de Traitement de Réponses

A 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*.

La suppression d'un message est bien évidemment conditionnée. En effet, elle tient compte des cas particuliers suivants :

- Supposons qu'un message M_1 envoyé, et que le délai d'attente de l'accusé de M_1 est dépassé, le message est alors ré-émis. Un premier accusé est traité et la cellule de M_1 signalée libre, et est désormais occupée par un message M_2. Par défaut, une case de données re-acquitte les messages, même s'ils sont traités. De ce fait, plusieurs accusès correspondant au même message peuvent être reçus. Le deuxième accusé est réceptionné, et tente de supprimer la cellule qu'occupe M_2. Pour prévenir contre ce cas nous avons renforcé la condition d'exécution d'une suppression logique d'un message de la *Liste Messages Non Encore Acquittés*. La condition de suppression d'une cellule *c* est conditionnée par la vérification de la conjonction (*c.libre* = '*n*') & (*c.estampille* = *accusé.estampille*).
- Supposons qu'un message M_1 envoyé, et que le délai d'attente de l'accusé de M_1 est dépassé plusieurs fois, jusqu'à ce que le message atteigne le nombre maximum de relances. Ceci entraîne une suppression du message M_1 de la *Liste Messages Non Encore Acquittés*, et son insertion dans la *Liste Messages Non Acquittés*. C'est après, qu'un des accusés parvienne. La conjonction (*c.libre* = '*n*') & (*c.estampille* = *accusé.estampille*) interdit au *Thread Traitement Rèponse* de supprimer logiquement le contenu de la cellule *c*. Par contre, le message M_1 doit être supprimé de la *Liste Messages Non Acquittés*. Cette dernière est parcourue séquentiellement, à partir de la fin, afin de satisfaire l'égalité entre l'estampille du message et l'estampille de l'accusé. Un parcours sans résultat signifie qu'un accusé a été déjà traité, et a supprimé le message de la *Liste Messages Non Acquittés*.

5.4.4.3 Thread de Ré-émission de Messages

Le *Thread de Ré-émission de Messages* vérifie les délais d'envoi des messages envoyés et non acquittés afin de les re-envoyer. Il exécute l'algorithme suivant,

Algorithme 5-1: Re-émission de messages niveau client.

Pour *i* : 0 .. *Crédit Envoi* Faire
  Si (*Liste Messages Non Encore Acquittés*[*i*].*Libre* = '*n*') Alors
    Si (🕒[5] - *Liste Messages Non Encore Acquittés*[*i*].*Temps Envoi* ≥ (1 + *Liste Messages Non Encore Acquittés*[*i*].*Nombre Relances*)*Délai Accusé*) & (*Liste Messages Non Encore Acquittés*[*i*].*Nombre Relances* < *Nombre Maximum Relances*) Alors
      Re-envoyer *Liste Messages Non Encore Acquittés*[*i*].*Message* à

[5] Le symbole 🕒 désigne l'horloge système, qui revoie l'heure système.

la case *Liste Messages Non Encore Acquittés*[*i*].*Destinataire*,
Liste Messages Non Encore Acquittés[*i*].*Nombre Relances* ++
Sinon Si (*Liste Messages Non Encore Acquittés*[*i*].*Nombre Relances* = *Nombre Maximum Relances*)
Alors
Supprimer logiquement la cellule *i* :
Liste Messages Non Encore Acquittés[*i*].*Libre* ← *'o'*,
Augmenter crédit envoi : *fenêtre*++,
Empiler le message dans *Liste Messages Non Acquittés*,
Fin Si
Fin Si
Dormir pendant *d* ms
Fin Pour

Notons que nous faisons endormir le *Thread de Ré-émission de Messages*, car sans cela les performances se dégradent. Ceci est dû à l'accès concurrent aux paramètres globaux par plusieurs *threads*. Le *Thread de Ré-émission de Messages* s'exécute ainsi périodiquement.

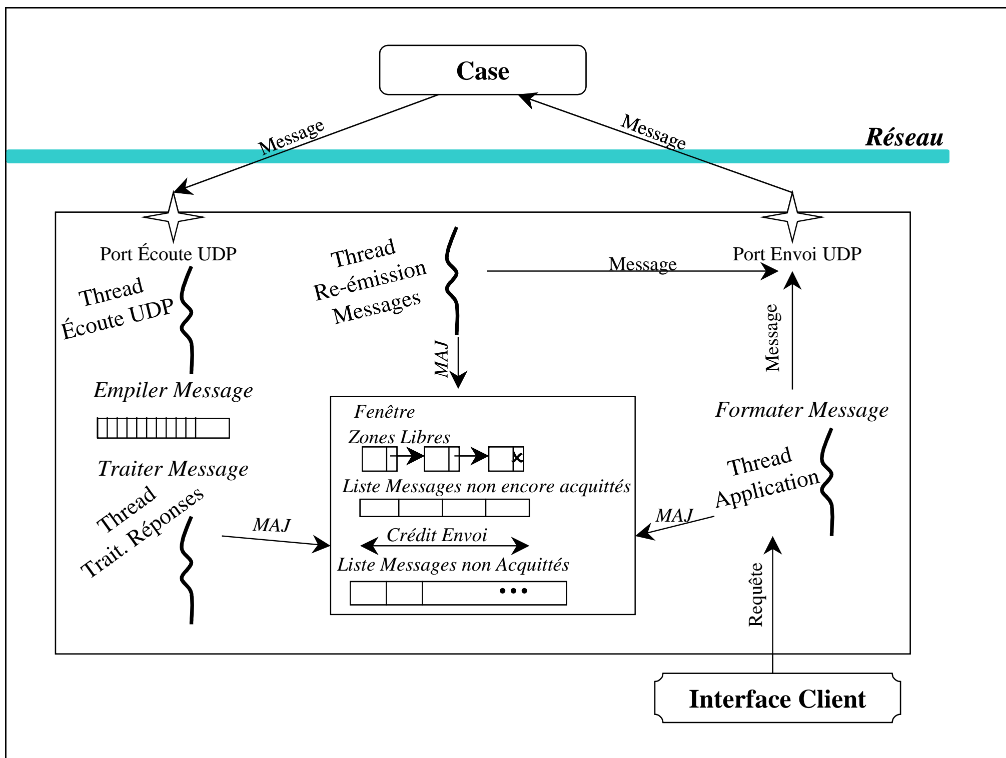
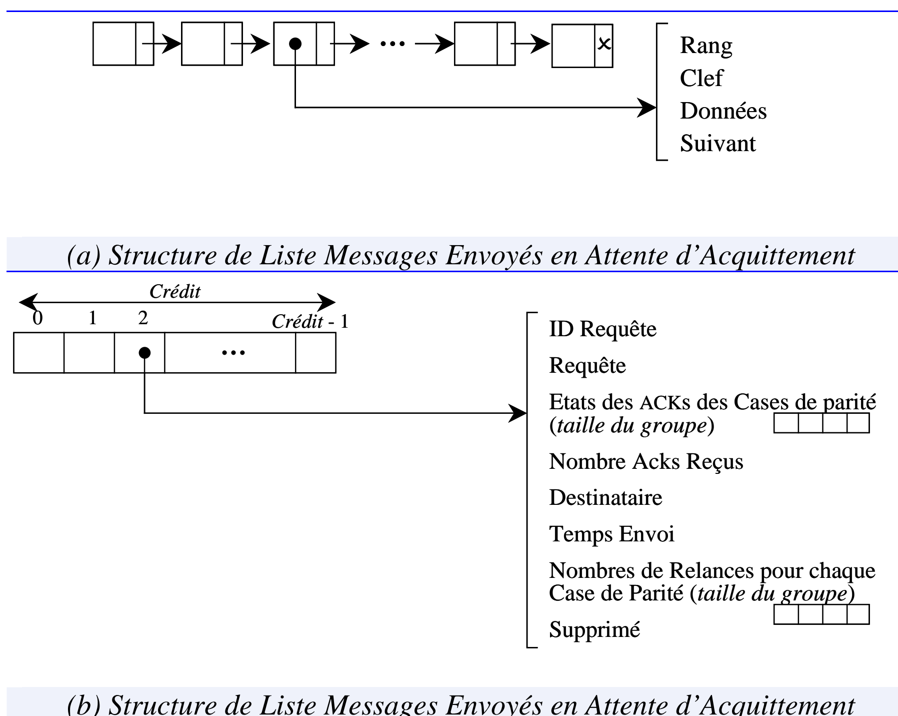


Figure 5-5: Architecture d'un client SDDS-Ack.

5.4.5 CFAM au Niveau d'une Case de Données

Notre stratégie de Contrôle de Flux et d'Acquittement de Messages (en abrégé CFAM) se dote, en plus de la structure proposée pour le CFAM entre un client et une case de données, d'une *File de Messages à Envoyer* et d'un *Thread d'Envoi de Messages*. Dans ce qui suit, nous décrivons le déroulement de CFAM.
A la réception d'une requête de mise à jour, insertion ou suppression, le *Thread de Travail* traitant la requête, insère le message de mise à jour à propager vers les cases de parité du groupe dans la structure: *File de Messages à Envoyer*.

Le *Thread d'Envoi de Messages* guette l'évènement *Existe Message à Envoyer*, signalant que la *File de Messages à Envoyer* n'est pas vide. Chaque fois qu'il dépile un message, il lui attribue un identifiant, l'envoie à l'ensemble des cases de parité du groupe et l'insère dans la *File des Messages en Attente d'Acquittement*. Ceci sachant que le *Thread d'Envoi de Messages* une fois son crédit d'envoi est épuisé, il arrête les envois.
Etant donné que les messages de mise à jour à propager vers les cases de parité contiennent le même champ servant pour la mise à jour, les structures *File de Messages à Envoyer* et *File des Messages en Attente d'Acquittement* sont optimales. En effet, le message n'est sauvé dans les files qu'une seule fois, et non *k* fois (*k* étant le niveau de disponibilité du groupe). A cet effet, nous supposons que tout message de la *File de Messages à Envoyer* est à envoyer à toutes les cases de parité du groupe. Par contre, une structure additionnelle est requise pour savoir si la case de parité *p* a accusé réception d'un message *M* ou non. Pour ce, un champ de taille *k* de booléens est réservé pour chaque message envoyé. Le message n'est alors supprimé que si tous ses destinataires accusent sa réception. Chaque file a la structure illustrée dans la Figure 5-6.



(a) Structure de Liste Messages Envoyés en Attente d'Acquittement

(b) Structure de Liste Messages Envoyés en Attente d'Acquittement

Figure 5-6: Structures d'acquittement niveau case de données.

Une case de parité acquitte le message de mise à jour en renvoyant à la case de données émettrice son niveau, l'identifiant du message, et le numéro cellule qu'occupe le message dans la *File des Messages en Attente d'Acquittement*.
Au niveau d'une case de données, à la réception d'un acquittement, un *Thread de Travail*, met à jour la *File des Messages en Attente d'Acquittement*. Si tous les acquittements attendus sont reçus, le message est supprimé, et le crédit d'envoi est incrémenté de 1.
La retransmission des messages est exécutée par le *Thread de Retransmission de Messages*. Ce dernier, analyse périodiquement chaque cellule non marquée supprimée de la *File des Messages en Attente d'Acquittement*, vérifie si le délai d'attente du message a expiré ou non, s'il a expiré et que le nombre maximum de retransmissions

n'est pas atteint, le message est retransmis aux cases de parité qui n'ont pas accusé réception du message en question. Enfin, dans le cas où, le message ne serait pas acquitté, toute la structure du message est copiée dans la liste des messages non-acquittés.
En ce qui concerne les cases de parité, celles-ci gardent trace des identifiants des derniers messages reçus de chaque case de données dans une liste à part (ang. *Taboo Lists*); et avant toute opération de mise à jour, elles vérifient que le message n'a pas été déjà reçu et traité. Dans le dernier cas, la case de parité re-acquitte le message auprès de la case de dernière.

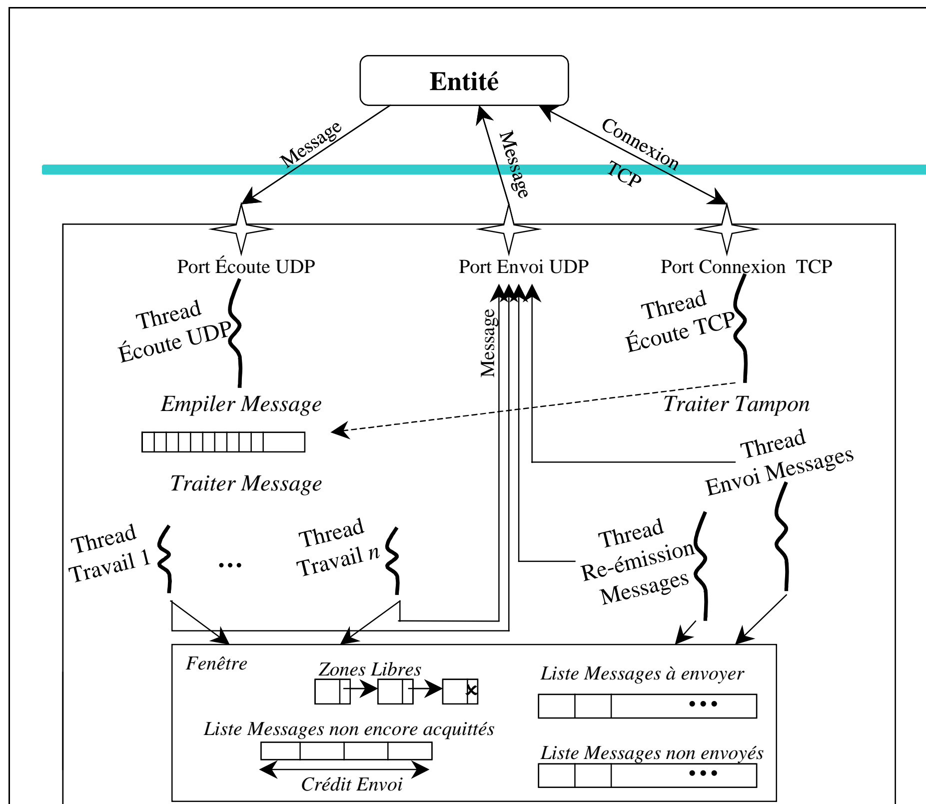


Figure 5-7: Architecture d'une Case SDDS-Ack.

Algorithme 5-2 : Re-émission de messages niveau case de données.

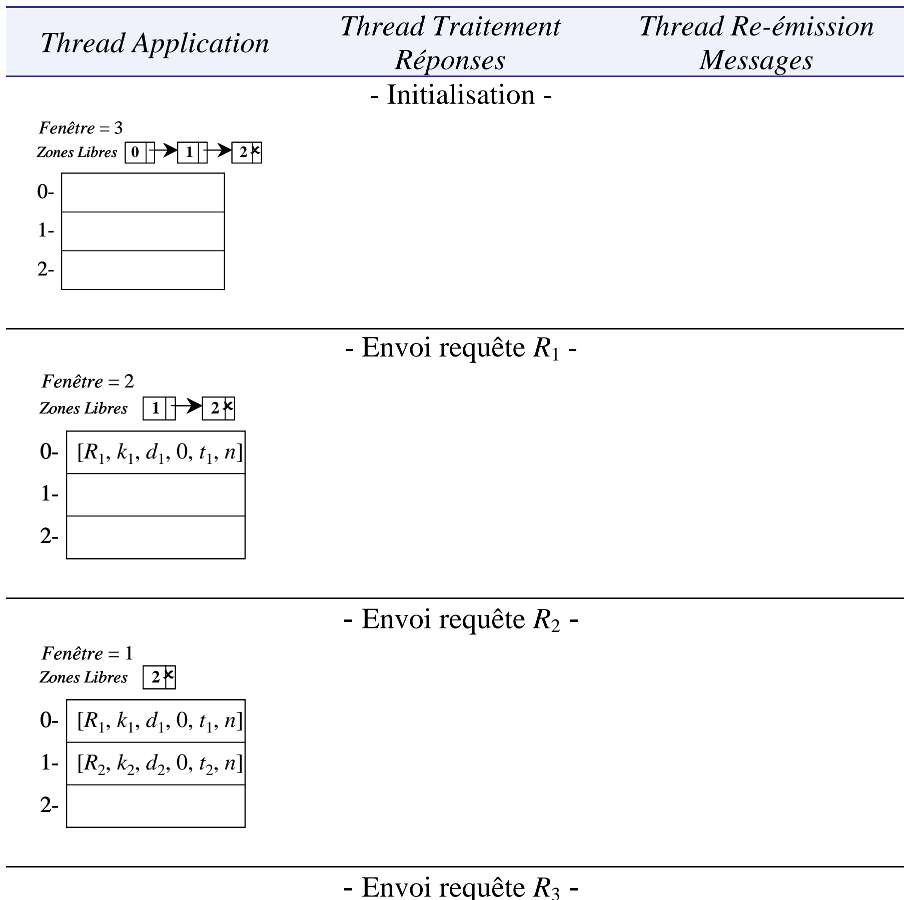
K : Niveau de disponibilité du groupe
Pour *i* : 0 .. *Crédit Envoi* Faire
  Si (*Liste Messages Non Encore Acquittés*[*i*].*Libre* = '*n*') Alors
    MaxRel ← 0
    Reçus ← 0
    Pour *p* : 1..*K* Faire
     Si (*Acks*[*p*] = 1) Alors *Reçus*++
     Sinon Si (🕒[6] - *Liste Messages Non Encore Acquittés*[*i*].*Temps Envoi* ≥ (1 + *Liste Messages Non Encore Acquittés*[*i*].*Nombre*

[6] Le symbole 🕒 désigne l'horloge système, qui revoie l'heure système.

Relances[*p*])**Délai Accusé*) & (*Liste Messages Non Encore Acquittés*[*i*].*Nombre Relances*[*p*] < *Nombre Maximum Relances*)
Alors
Re-envoyer *Liste Messages Non Encore Acquittés*[*i*].*Requête* à la case *Liste Messages Non Encore Acquittés*[*i*].*Destinataire*,
Liste Messages Non Encore Acquittés[*i*].*Nombre Relances*[*p*] ++
Sinon Si (*Liste Messages Non Encore Acquittés*[*i*].*Nombre Relances*[*p*] = *Nombre Maximum Relances*) Alors *MaxRel*++
Fin Si
Fin Pour
Si (*K* – *Reçus* = *MaxRel*) Alors
Supprimer logiquement la cellule *i*,
Liste Messages Non Encore Acquittés[*i*].*Libre* ← '*o*',
Augmenter crédit envoi: *fenêtre*++,
Empiler le message dans *Liste Messages Non Acquittés*,
Fin Si
Fin Si
Dormir pendant *d* ms
Fin Pour

5.4.6 Exemple

Pour comprendre la stratégie d'acquittement proposée, considérons l'exemple illustré dans la Figure 5-8. Ce dernier illustre la stratégie d'acquittement au niveau d'un client qui envoie des requêtes à destination d'une case.



Fenêtre = 0
Zones Libres [×]

0-	$[R_1, k_1, d_1, 0, t_1, n]$
1-	$[R_2, k_2, d_2, 0, t_2, n]$
2-	$[R_3, k_3, d_3, 0, t_3, n]$

- Réception accusé requête R_3-

Fenêtre = 1
Zones Libres [2|×]

0-	$[R_1, k_1, d_1, 0, t_1, n]$
1-	$[R_2, k_2, d_2, 0, t_2, n]$
2-	$[R_3, k_3, d_3, 0, t_3, o]$

- Réception accusé requête R_1-

Fenêtre = 2
Zones Libres [2|] → [0|×]

0-	$[R_1, k_1, d_1, 0, t_1, o]$
1-	$[R_2, k_2, d_2, 0, t_2, n]$
2-	$[R_3, k_3, d_3, 0, t_3, o]$

- Envoi requête R_4 -

Fenêtre = 1
Zones Libres [0|×]

0-	$[R_1, k_1, d_1, 0, t_1, o]$
1-	$[R_2, k_2, d_2, 0, t_2, n]$
2-	$[R_4, k_4, d_4, 0, t_4, n]$

- Envoi requête R_5 -

Fenêtre = 0
Zones Libres [×]

0-	$[R_5, k_5, d_5, 0, t_5, n]$
1-	$[R_2, k_2, d_2, 0, t_2, n]$
2-	$[R_4, k_4, d_4, 0, t_4, n]$

- (🕒 - $t_2 \geq$ *Délai Accusé*) -

0-	$[R_5, k_5, d_5, 0, t_5, n]$
1-	$[R_2, k_2, d_2, 1, t_2, n]$
2-	$[R_4, k_4, d_4, 0, t_4, n]$

- Réception accusé requête R_4 -

Fenêtre = 1
Zones Libres [2|×]

0-	$[R_5, k_5, d_5, 0, t_5, n]$
1-	$[R_2, k_2, d_2, 1, t_2, n]$
2-	$[R_4, k_4, d_4, 0, t_4, o]$

- (🕒 - $t_5 \geq$ *Délai Accusé*) & (🕒 - $t_2 \geq$ 2**DélaiAccusé*) -

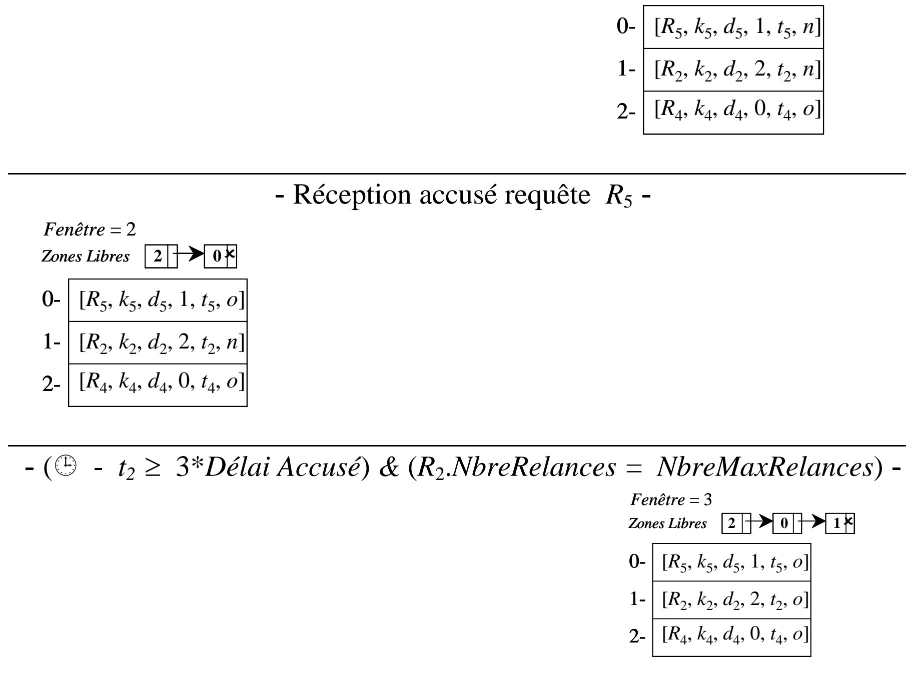


Figure 5-8: Exemple de déroulement de l'algorithme d'acquittement.

L'exemple de la Figure 5-8 montre l'envoi de cinq requêtes, et le déroulement de l'algorithme d'acquittement entre les trois thread, en conséquence de l'événement *Envoi message*, *Réception accusé*, *Délai dépassé* ou *Nombre max relances atteint*. Les trois *threads* mettent à jour les paramètres *Fenêtre*, *Zones Libres* et *Liste Messages Non Encore Acquittés*. Les paramètres *Crédit Envoi* et *Nombre Max de Relances* sont initialisés respectivement à 3 et 2.

5.5 Architecture SDDS-IP@

L'architecture baptisée *SDDS-IP@* a été proposée dans le but de supprimer la table d'adresses statique dans le programme. Dans les architectures précédentes, chaque case, détient une copie d'une table d'adresses statique. Cette dernière renseigne sur l'adresse IP de la machine hôte et le port d'écoute UDP de chaque case.

A partir du *port d'écoute UDP*, nous pouvons déduire le *port d'envoi UDP* étant le *port d'écoute UDP*+1, et le *port de connexion TCP* étant le *port d'écoute UDP*+2.

Les adresses IP des hôtes étaient saisies en mode programme. Une fois le programme compilé, une table d'adresses statique est générée, et est diffusée à toutes cases.

5.5.1 Structures d'adressage

Dans la nouvelle architecture nous avons uniformisé le calcul des ports, qui dépendent du numéro de la case en question. En effet, le *port d'écoute UDP* est désormais défini en fonction du numéro de case. Ainsi, une case de numéro *c*, a pour *port d'écoute UDP* : 6000+(3**c*), a pour *port d'envoi UDP* : 6000+(3**c*)+1, et a pour *port de connexion TCP/IP* : 6000+(3**c*)+2.

Le problème de résolution des ports d'*écoute/ envoi UDP* et *connexion TCP/IP* reposant sur la connaissance des numéros des cases, en plus de l'algorithme d'adressage LH* reposant sur les numéros logiques des cases de données posent une nouvelle problématique. En effet, nous nous posons la question, quels numéros faut-il attribuer aux cases de parité ?

Ceci nous a amené à faire la distinction entre *numéro entité* et *numéro logique*. Le coordinateur est maître de l'attribution des numéros d'entités, puisque c'est lui qui contrôle les éclatements et de la haute disponibilité. Chaque case de données et de parité d'un fichier $LH*_{RS}$ possède un numéro d'entité. Ce dernier permet le calcul des paramètres *ports d'écoute/ envoi UDP* et *connexion TCP/IP*.

Notons que les numéros logiques sont attribués exclusivement aux cases de données. Ils sont obligatoires, car ils constituent le principe d'adressage LH*. La correspondance entre les couples de numéros pour les cases de données est établie par des structures.

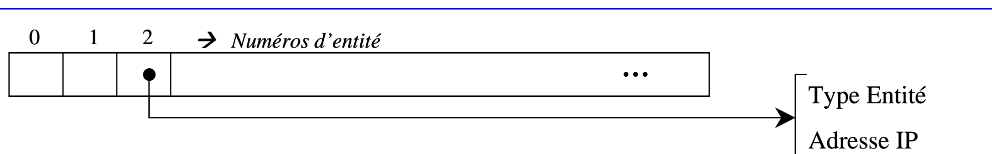
Ainsi, nous avons remplacé la table statique des adresses par des structures adéquates spécifiques aux besoins de connaissances d'adresse de chaque entité. En effet, à titre d'exemple : une case de données n'est pas censée savoir l'adresse de l'hôte qui héberge la $i^{éme}$ case de parité d'un groupe auquel elle n'appartient pas.

Dans ce qui suit, nous décrivons la structure d'adressage correspondante au coordinateur, à une case de données, une case de parité, et à un client.

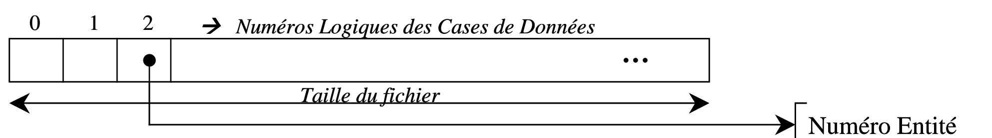
5.5.1.1 Coordinateur

Le tableau *Adresses-des-Entités* renseigne sur le type de chaque entité : case de données, case de parité ou client, et sur l'adresse IP de la machine où l'entité en question se trouve. L'indice de chaque cellule constitue le numéro de l'entité, et donc permet de calculer les paramètres : *port écoute UDP*, *port envoi UDP* et *port connexion TCP/IP*.

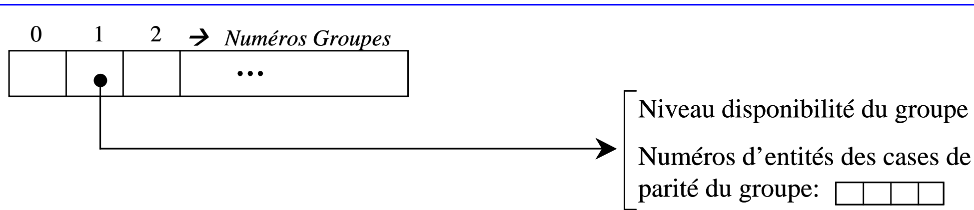
La correspondance entre les numéros logiques des cases de données et les numéros des entités est assurée par la table *Numéros-Cases-de-Données*. L'indice de chaque cellule de la table *Numéros-Cases-de-Données* est le numéro logique d'une case de données, alors que le contenu de la cellule est le numéro d'entité. Il faudrait faire la correspondance avec la structure *Adresses-des-Entités* pour avoir l'adresse IP d'une case de données.



(*a*) Adresses des Entités



(*b*) Numéros des Cases de Données



(*c*) Informations sur les groupes de parité

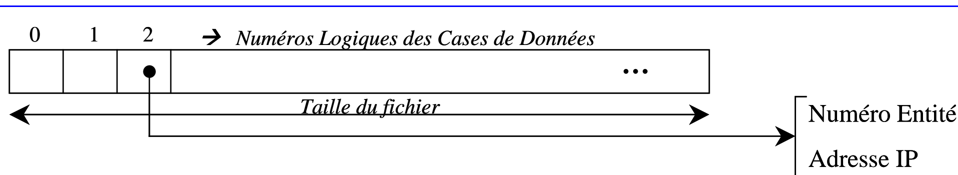
Figure 5-9: Structures d'adressage du Coordinateur.

La correspondance entre les numéros logiques des cases de parité et les numéros des entités est assurée par la table *InfoGroupe*. Tel que, l'indice de chaque cellule est le numéro d'un groupe. Chaque cellule possède deux champs. Le premier renseigne sur le niveau de disponibilité du groupe, alors que le deuxième est une liste contenant les numéros d'entité des cases de parité du groupe. Il faudrait faire la correspondance avec la structure *Adresses-des-Entités* pour avoir l'adresse IP d'une case de parité.

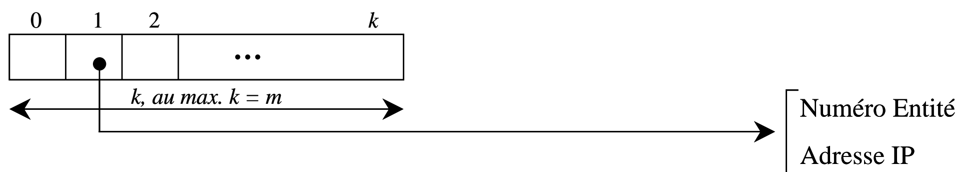
5.5.1.2 Case de Données

Les besoins d'une case de données sont différents de ceux du coordinateur. Pour ce, nous avons conçu deux structures *Adresses-Cases-Données* et *Adresses-Mes-Cases-de-Parité* répondant aux besoins d'une case de données.

La structure *Adresses-Cases-Données* est un tableau, tel que l'indice de chaque cellule est le numéro logique d'une case de données. Chaque cellule possède deux champs. Le premier renseigne sur le numéro d'entité de la case de données, alors que le deuxième renseigne sur l'adresse IP de la case de données. La connaissance des adresses IP des cases de données est utile pour le renvoi de requêtes.



(*a*) Adresses des Cases de Données



(*b*) Adresses des Cases de Parité du groupe auquel appartient la case de données

Figure 5-10: Structures d'adressage d'une case de données.

La structure *Adresses-Mes-Cases-de-Parité* renseigne sur les cases de parité du groupe auquel la case de données appartient. Chaque cellule possède deux champs. Le premier renseigne sur le numéro d'entité de la case de parité, alors que le deuxième

renseigne sur l'adresse IP de la case de parité. Cette structure est utile pour la propagation des mises à jour vers les cases de parité.

L'adresse du coordinateur, étant la case de données de numéro logique 0, se trouve dans la première cellule de la table *Adresses-Cases-Données*

5.5.1.3 Case de Parité

Les besoins d'une case de parité se limitent à la connaissance des adresses IP et des numéros d'entité des cases de données du groupe de parité, auquel elle est rattachée. Pour ce, la structure *Adresses-Mes-Cases-Données* a été conçue.

Chaque cellule de la structure *Adresses-Mes-Cases-Données* possède deux champs. Le premier renseigne sur le numéro d'entité de la case de données, alors que le deuxième renseigne sur l'adresse IP de la case de données.

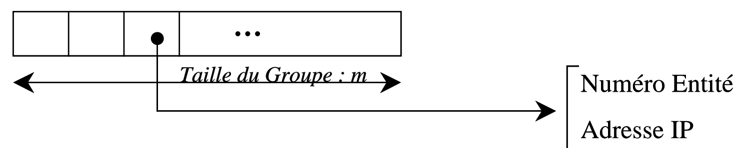


Figure 5-11: Structure d'adressage d'une Case de Parité.

5.5.1.4 Client

Le client ignore l'existence de cases de parité et la structure de groupage à laquelle les cases de données et de parité obéissent. Les besoins d'un client en terme de connaissance d'adresses, se limitent à la connaissance des adresses des cases de données. La structure *Adresses-Cases-de-Données* est un tableau, tel que l'indice de chaque cellule est le numéro logique d'une case de données. Chaque cellule possède deux champs. Le premier renseigne sur le numéro d'entité de la case de données, alors que le deuxième renseigne sur l'adresse IP de la case de données.

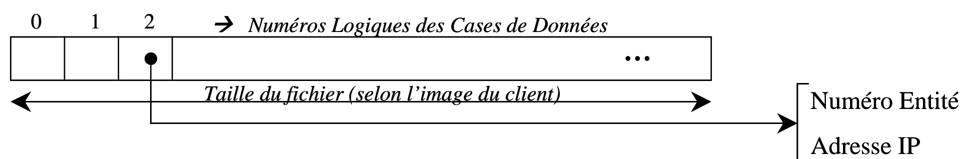


Figure 5-12: Structure d'adressage d'un client.

5.5.2 Groupes Multicast

Nous avons crée deux groupes multicast, notamment le *groupe des cases de données vierges* et le *groupe des cases de parité vierges*. Chaque groupe est caractérisé par une *adresse multicast* et un *port d'écoute multicast*.

Pour le *groupe des cases de données vierges* l'*adresse multicast* choisie est '233.1.1.1'. Les cases de données lancées se mettent à l'écoute de messages parvenant à l'adresse multicast '233.1.1.1' sur le *port d'écoute multicast* '10000'.

Pour le *groupe des cases de parité vierges* l'*adresse multicast* choisie est '233.1.1.2'. Les cases de parité lancées se mettent à l'écoute de messages parvenant à l'adresse multicast '233.1.1.2' sur le *port d'écoute multicast* '10001'.

A son initialisation, une case de données (resp. de parité) se connecte au *groupe des cases de données (resp. de parité) vierges*. Ceci est conditionné, car deux cases initialisés sur la même machine ne peuvent pas faire l'écoute multicast sur le même port d'écoute multicast. Ainsi, la deuxième case lancée se met en attente de la libération du port d'écoute multicast. La libération du port d'écoute est faite dès que la case réquisitionnant le port multicast quitte le groupe multicast.

5.5.3 Architecture

L'architecture *SDDS-IP@*, ajoute deux threads: le *Thread d'Ecoute Multicast* et le *Thread de Traitement Multicast*. Dans ce qui suit, nous décrivons le mode de fonctionnement de chacun des threads.

1. *Thread d'Ecoute Multicast*

   Le *Thread Ecoute Multicast* reste à l'affût des messages entrant sur le *port d'écoute multicast*. Il empile alors les messages reçus dans la file de requêtes multicast reçues, et signale la présence de requêtes multicast à traiter au *Thread de Traitement Multicast* par l'événement *ExisteRequêteMulticast*.

2. *Thread de Traitement Multicast*

   Le *Thread de Traitement Multicast* reste à l'affût d'un signal de l'événement *ExisteRequêteMulticast*, pour dépiler une requête, l'analyser et la traiter.

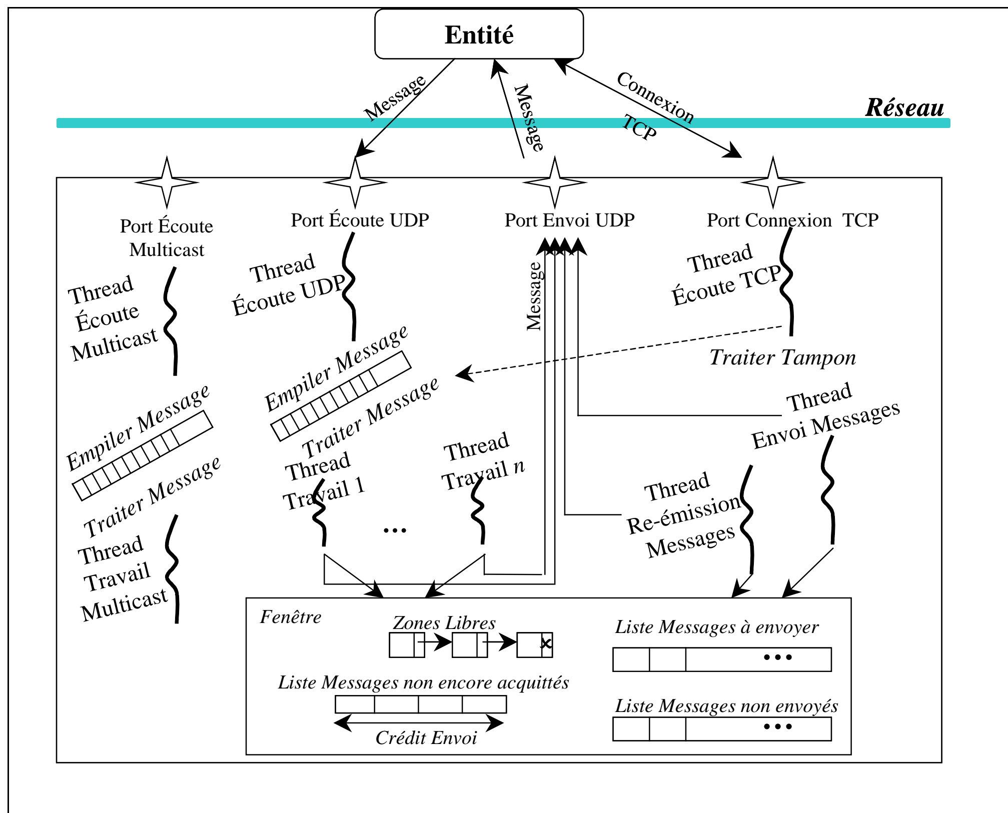


Figure 5-13: Architecture SDDS-IP@ d'une case.

5.6 Architecture Fonctionnelle

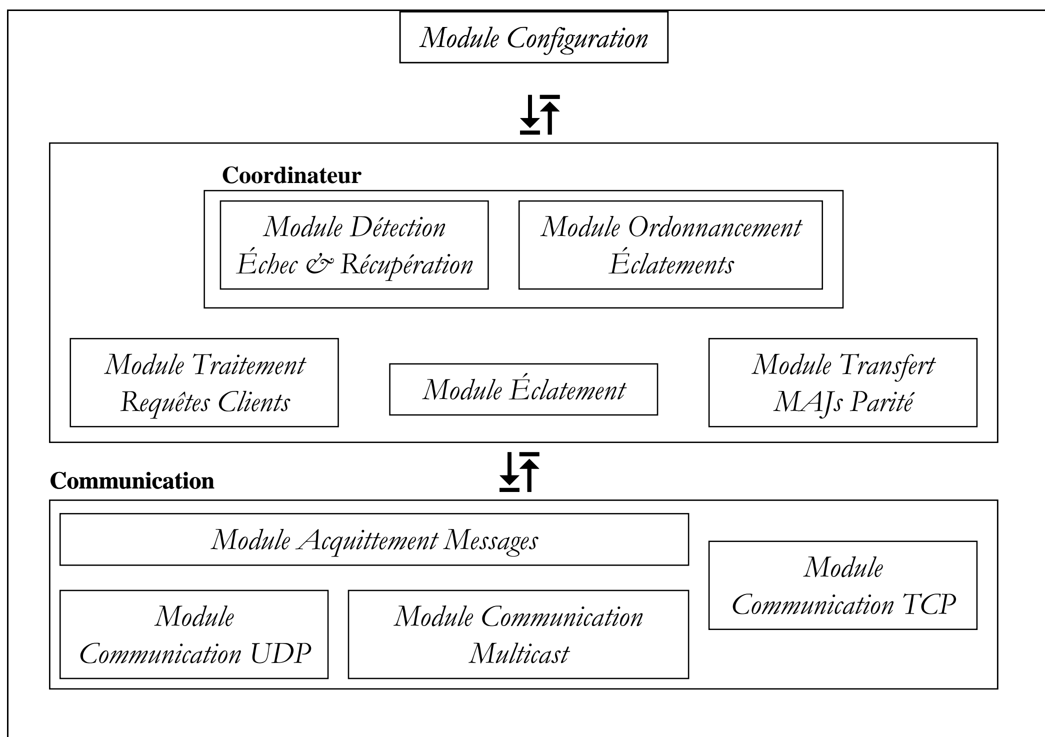


Figure 5-14: Architecture Fonctionnelle d'une Case de Données.

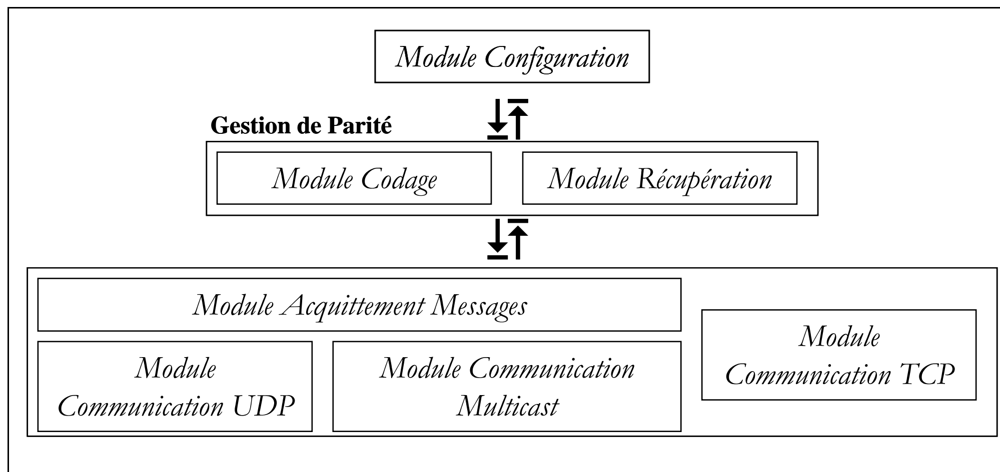


Figure 5-15: Architecture Fonctionnelle d'une Case de Parité.

5.7 Conclusion

La conception d'un scénario $LH*_{RS}$ est différente dans chacune des architectures. Le chapitre suivant décrit les différents scénarii dans chacune des architectures décrites tout au long de ce chapitre.

CHAPITRE

6 ARCHITECTURE OPERATIONNELLE DU GESTIONNAIRE LH*$_{RS}$

6.1 Introduction

Le présent chapitre décrit l'architecture fonctionnelle de notre gestionnaire LH*$_{RS}$. Nous décrivons les interactions de la couche fonctionnelle de LH*$_{RS}$ avec l'architecture des cases et leurs organisations internes (voir Figure 6-1). Le modèle de conception choisi est l'illustration par scénario de chaque fonctionnalité. Un scénario définit les interactions entre les différentes entités : client, coordinateur, case de données, case de parité et cases de secours, afin de réaliser un traitement. Dans ce qui suit, nous décrivons les fonctionnalités suivantes de LH*$_{RS}$:

* Manipulations d'enregistrements par le client: insertion, suppression, mise à jour, la recherche par clé d'un enregistrement de données dans un fichier LH*$_{RS}$.
* Eclatement d'une case de données LH*$_{RS}$.
* Augmentation de la disponibilité d'un groupe par l'ajout d'une case de parité.
* Récupération d'un enregistrement.
* Récupération d'une ou plusieurs cases.

Nous exposons pour chaque scénario différentes solutions que nous proposons et une analyse de la complexité de chaque solution. Nous proposons également d'éventuelles améliorations.

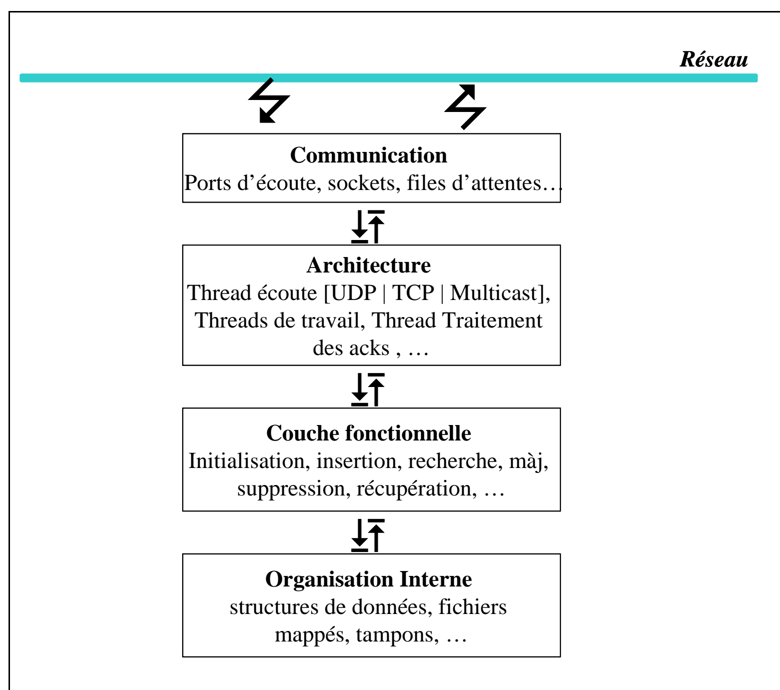


Figure 6-1: Architecture globale de SDDS-2000.

Tout au long du présent chapitre, nous adoptons la terminologie suivante:

Symbole	*Description*
Ed	Enregistrement de données
Ep	Enregistrement de parité
m	taille d'un groupe
k	niveau de disponibilité d'un groupe *g*
G	Matrice génératrice
l	niveau de la case de parité dans la matrice génératrice G
G^l	la $1^{éme}$ colonne de *G* – Niveau d'une case de parité
G_i	la $i^{éme}$ colonne de *G* – correspond à une case de données
G_i^l	Coefficient se trouvant dans cellule (*i*,*l*) de *G*
t	taille des attributs non-clé d'un enregistrement, exprimée en nombre de symboles
%, mod	opérateur modulo
+, -, ^, ⊕	Opération dans un champ de Galois ⇔ ou-exclusif, XOR
*	multiplication dans un champ de Galois
x + = y	$\Leftrightarrow x \leftarrow x + y$, idem pour $x -= y$
!=	$\Leftrightarrow \neq$

6.2 Scénarios de Manipulations Client

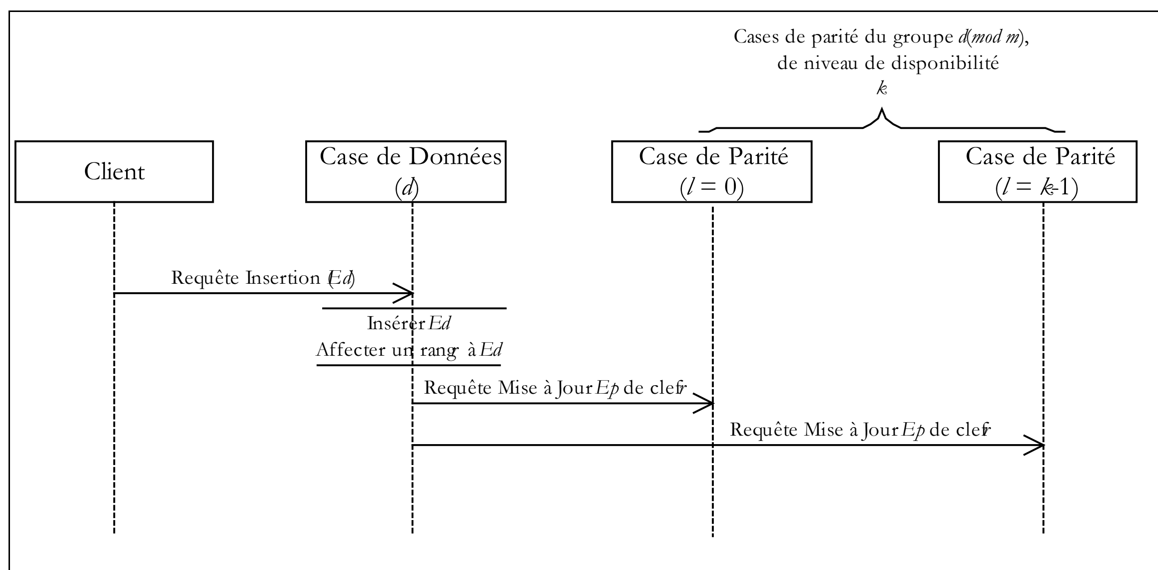
Le client manipule les enregistrements de données. Une *manipulation client* peut être une insertion, une suppression, une mise à jour et une recherche par clé d'un enregistrement de données *Ed* appartenant à une case de données *d*. Dans les trois premiers cas, la manipulation se répercute sur *k* enregistrements de parité, tel que *k* désigne le niveau de disponibilité du groupe de cases auquel *d* appartient. Les *k* enregistrements de parité ont pour clé : le rang occupé par *Ed* dans *d*.

Selon l'algorithme LH*, le client, sur la base de son image du fichier, calcule l'adresse de la case à laquelle la requête doit être envoyée. Une fois l'adresse déterminée, le client envoie la requête par UDP à la case. Cette dernière traite ou transfère la requête à une autre case. La réponse ou l'accusé d'une requête client peut être accompagnée d'un message d'ajustement de l'image du client, si la requête a subi un transfert.

Dans ce qui suit, les scénarii illustrés supposent que l'image du client est correcte, donc la requête atteint la case de données cible.

6.2.1 Insertion d'un Enregistrement

Le Scénario 6-1 illustre l'insertion d'un enregistrement *Ed* dans *d*. La requête d'insertion du client se répercute sur toutes les cases de parité du groupe auquel la case de données *d* appartient. A cette fin, *d* envoie aux cases de parité les messages d'insertion/ mise à jour. Remarquons que nous ne montrons pas qu'une partie du déroulement, les acquittements sont omis.



Scénario 6-1: Insertion d'un Enregistrement de Données.

Le message d'insertion d'un enregistrement de parité comporte les champs suivants:

d : adresse logique de la case de données émettrice,
Ed. Clé : clé de l'enregistrement de données *Ed*,
Ed. Attributs : champs des attributs non-clef de l'enregistrement de données *Ed*,
r : le rang qu'occupe *Ed* dans la case *d*…

A la réception d'un message d'insertion, la case de parité de niveau l commence par rechercher l'enregistrement de parité *Ep* de clé r. Ainsi, nous distinguons deux cas selon un enregistrement de parité de clé r existe ou non. Le calcul du champ de parité de *Ep* s'effectue comme suit, *Ed. Attributs* est considéré en tant que chaîne de symboles, et il s'agit d'itérer sur le nombre de symboles de *Ed.Attributs*, tel qu'à chaque itération un symbole de *Ep.Parité* est mis à jour.

Algorithme 6-1: Insertion d'un enreg. de données par une case de parité.

Si *Ep* n'existe pas Alors
  Créer *Ep* de clé *rang*,
  Initialiser *Ep. Liste des clés* à –1,
  Ep. Liste des clés [d mod m] ← *Ed. Clé*,
  Ep. Parité ← *Ed. Attributs* * G^l,
    p_i : $i^{éme}$ symbole de *Ep.Parité*
    d_i : $i^{éme}$ symbole de *Ed*
    Cas de codage RS
      Pour i : 1..t, p_i ← d_i * G [d mod m] [l]
    Cas de codage XOR
      Pour i : 1..t, p_i ← d_i
  Incrémenter le nombre d'enregistrements de la case.
Sinon Il s'agit alors de mettre à jour *Ep*,
  Ep. Liste des clés [d mod m] ← *Ed. Clé*,
  Ep. Parité ^ = *Ed.Attributs* * G^l.
    p_i : $i^{éme}$ symbole de *Ep.Parité*
    d_i : $i^{éme}$ symbole de *Ed*
    Cas de codage RS
      Pour i : 1..t, p_i ^= d_i * G [d mod m] [l]
    Cas de codage XOR
      Pour i : 1..t, p_i ^= d_i
Fin Si

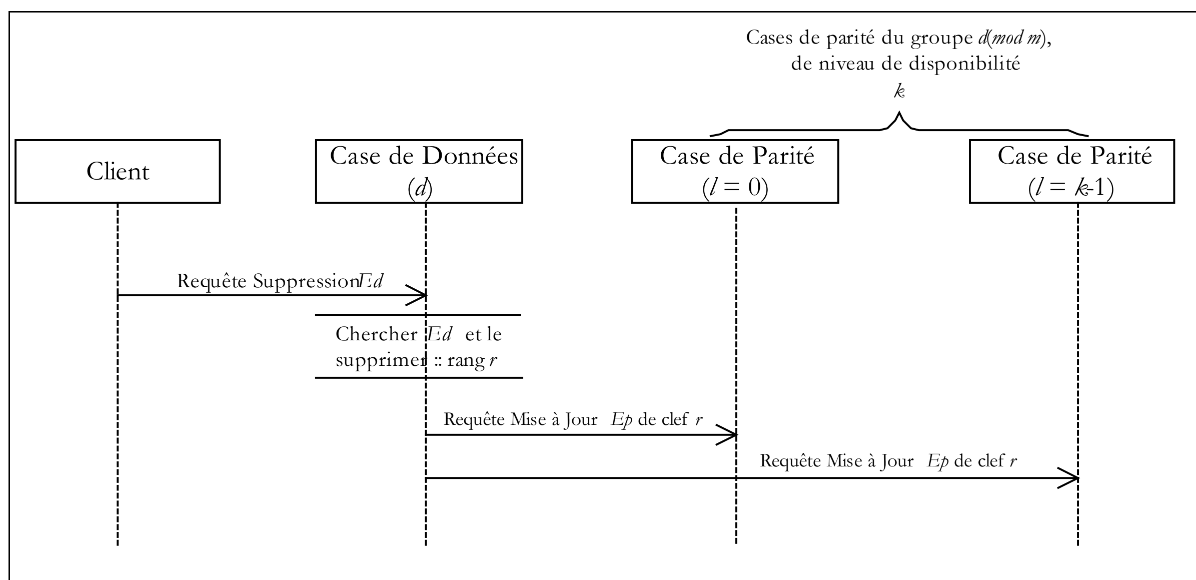
Notons que:

* En cas de codage RS, la mise à jour d'un symbole nécessite, une opération XOR et une opération de multiplication GF. Par conséquent, la mise à jour d'un champ de parité est de t opérations XORs et t opérations de multiplication GF.

* En cas de codage XOR, la mise à jour d'un champ de parité est de t opérations XORs.

6.2.2 Suppression d'un Enregistrement

Le Scénario 6-2 illustre la suppression d'un enregistrement de données *Ed*, appartenant à la case de données d. La requête de suppression se répercute sur l'ensemble des cases de parité du groupe auquel la case de données d appartient.



Scénario 6-2: Suppression d'un enregistrement de données.

Le message de suppression comporte les champs suivants:

> *d* : adresse logique de la case de données, à laquelle *Ed* appartient,
> *Ed.clé* : clé de l'enregistrement de données *Ed*,
> *Ed.attributs* : champs attributs non-clef de l'enregistrement de données *Ed*,
> *r*: rang occupé par *Ed* dans la case *d*…

A la réception d'un message de suppression d'un enregistrement de données *Ed*, une case de données *d* répercute cette suppression sur les cases de parité du groupe, auquel elle appartient. La case de parité, de niveau *l*, commence par chercher l'enregistrement de parité de clé *rang*. Une fois ce dernier est trouvé, il est mis à jour selon l'Algorithme 6-2.

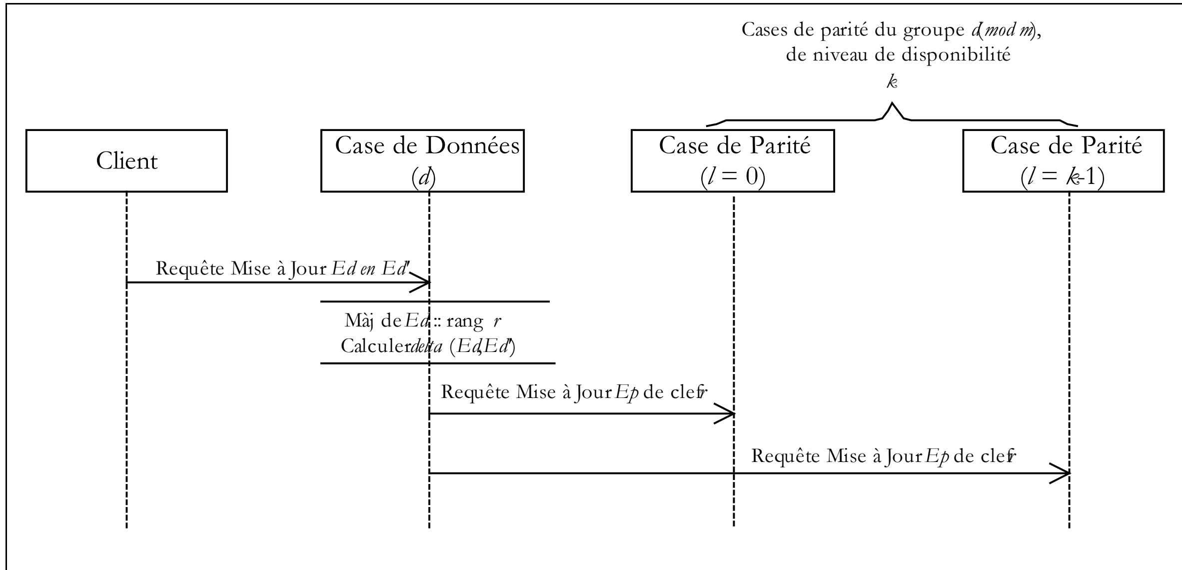
Algorithme 6-2: Suppression d'un enreg. de données par une case de parité.

p_i : $i^{éme}$ symbole de *Ep.Parité*
d_i : $i^{éme}$ symbole de *Ed*
Ep. Liste des clés [*d* mod *m*] ← -1, *Ed* devient factice,
Cas de codage RS
    Pour i : 1..t, p_i ^= d_i * G [d mod m] [l]
Cas de codage XOR
    Pour i : 1..t, p_i ^= d_i

Outre la mise à jour de l'enregistrement de parité, quand les *m* éléments de *Ep. Liste des clés* deviennent –1, le nombre d'enregistrements dans la case est décrémenté. *Ep* est marqué supprimé logiquement pour éviter des confusions en cas de récupération.

6.2.3 Mise à Jour d'un Enregistrement

Le Scénario 6-3 illustre la mise à jour d'un enregistrement de données *Ed* appartenant à la case de données *d*. La requête de mise à jour se répercute sur l'ensemble des cases de parité du groupe auquel la case de données *d* appartient.



Scénario 6-3: Mise à jour d'un enregistrement de données.

Le message de mise à jour envoyé par le client comporte les champs suivants:

> *d* : adresse logique de la case de données, à laquelle *Ed* appartient,
> *Ed*: champs attributs de l'enregistrement de données *Ed*,
> *rang* : rang occupé par *Ed* dans la case *d*…

A la réception d'un message de mise à jour, la case de parité de niveau *l* recherche l'enregistrement de parité de clé *r*, étant *Ep*. Puis, met à jour *Ep. Parité*. Intuitivement, il s'agit de supprimer les données relatives aux anciennes valeurs de *Ed.Attributs*, puis insérer les nouvelles valeurs de *Ed.Attributs'*.

Algorithme 6-3: Mise à jour d'un enregistrement de données par une case de parité.

p_i : $i^{éme}$ symbole de *Ep.Parité*
d_i : $i^{éme}$ symbole de *Ed.Attributs*
d'_i : $i^{éme}$ symbole de *Ed.Attributs'*
Cas de codage RS
    Pour i : 1..t, p_i ^= d'_i * G [d mod m] [l] ^ d'_i * G [d mod m] [l]
Cas de codage XOR
    Pour i : 1..t, p_i ^= d'_i ^ d_i

Ainsi, en cas de codage RS, une mise à jour nécessite deux opérations XOR et deux multiplications GF par symbole, donc 2*t opérations XORs et 2*t opérations de multiplication GF. Ceci mis à part, le fait qu'il faut envoyer à la case de parité l'ancienne et la nouvelle valeur de champs attributs non-clef de *Ed*. l'opération de mise à jour est optimisée. En effet, dans une première étape, nous calculons le delta enregistrement $\triangle Ed$, égal à *Ed. Attributs'* ^ *Ed. Attributs*. Le $\triangle Ed$ est calculé en t opérations XORs. Seul le delta enregistrement est envoyé aux k cases de parité. Chaque case de parité de niveau l met à jour son champ de parité, en le XORant au résultat de la multiplication de $\triangle Ed$ par le $G^{l}_{d \bmod m}$. Ceci coûte t opérations XORs et t

opérations de GF-multiplication, pour le total de 2**t* opérations XORs plus *t* opérations de GF-multiplication. Ce mode de mise à jour permet d'économiser *t* opérations de CG-multiplication.

Algorithme 6-4: Mise à jour optimisée d'un enregistrement de données par une case de parité.

p_i : $i^{éme}$ symbole de *Ep.Parité*
d_i : $i^{éme}$ symbole de $\triangle Ed$
Cas de codage RS
    Pour i : 1..t, p_i ^= d_i * G [d mod m] [l]
Cas de codage XOR
    Pour i : 1..t, p_i ^= d_i

Remarques

Les scénarios décrits supposent que le champ données et le champ parité ont la même taille, et que les enregistrements sont de taille fixe. Dans le cas contraire, la taille d'un enregistrement de parité est égale à la taille du plus grand enregistrement de données. Par conséquent, l'insertion, suppression ou mise à jour d'un enregistrement de données n'entraîne que la mise à jour de seulement *t* symboles des *k* enregistrements de parité, tel que *t* désigne la taille de l'enregistrement de données en question.

Les scénarios n'illustrent pas les acquittements.

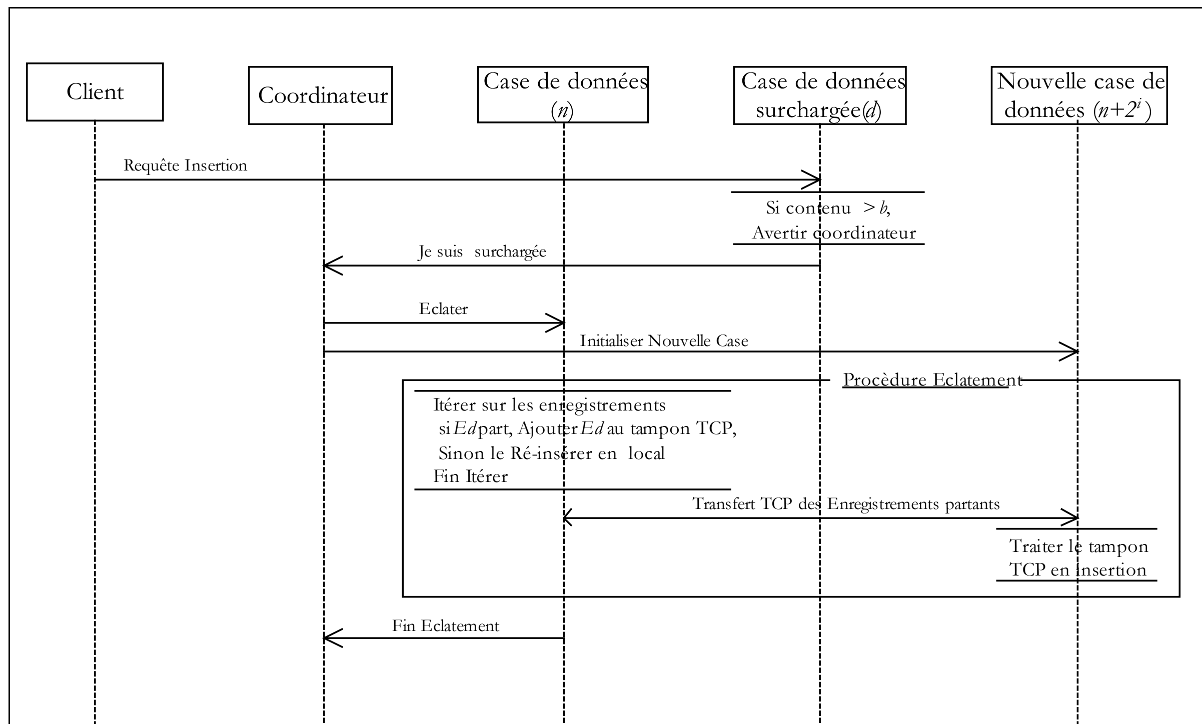
6.3 Scénario d'Eclatement d'une Case de Données $LH*_{RS}$

La procédure d'éclatement d'une case de données $LH*_{RS}$ se fonde sur celle de $LH*_{LH}$, [B00]. Elle est néanmoins bien plus complexe, impliquant les *k* cases de parité. Dans un premier temps, nous rappelons par un scénario approprié l'éclatement d'une case $LH*_{LH}$. Puis, nous discutons le premier scénario d'éclatement $LH*_{RS}$ proposé par Ljungström [L00]. Enfin, nous présentons notre scénario d'éclatement $LH*_{RS}$.

6.3.1 Eclatement d'une Case de Données $LH*_{LH}$

Rappelons qu'un éclatement $LH*_{LH}$, est déclenché par une surcharge au niveau d'une case de données, et est suivi par un éclatement d'une case de données *n*, désignée par le coordinateur (§2.6.1.1). Le Scénario 6-4 montre le déroulement de l'éclatement d'une case de données organisée en $LH*_{LH}$. Nous avons amélioré le scénario au niveau du transfert des enregistrements de données vers la nouvelle case.

Notons que dans le scénario d'éclatement implanté par [B00], le transfert des enregistrements de données de la case de données *n* à la case de données '$n + 2^i$', utilise bien TCP/IP, mais l'envoi des enregistrements de données se fait enregistrement par enregistrement. Coté récepteur, le traitement en insertion des enregistrements n'est fait qu'une fois tous les enregistrements de données sont reçus. Nous avons changé le scénario du coté de la case qui éclate. Celle-ci désormais envoie la totalité des enregistrements à transmettre en une seule fois.

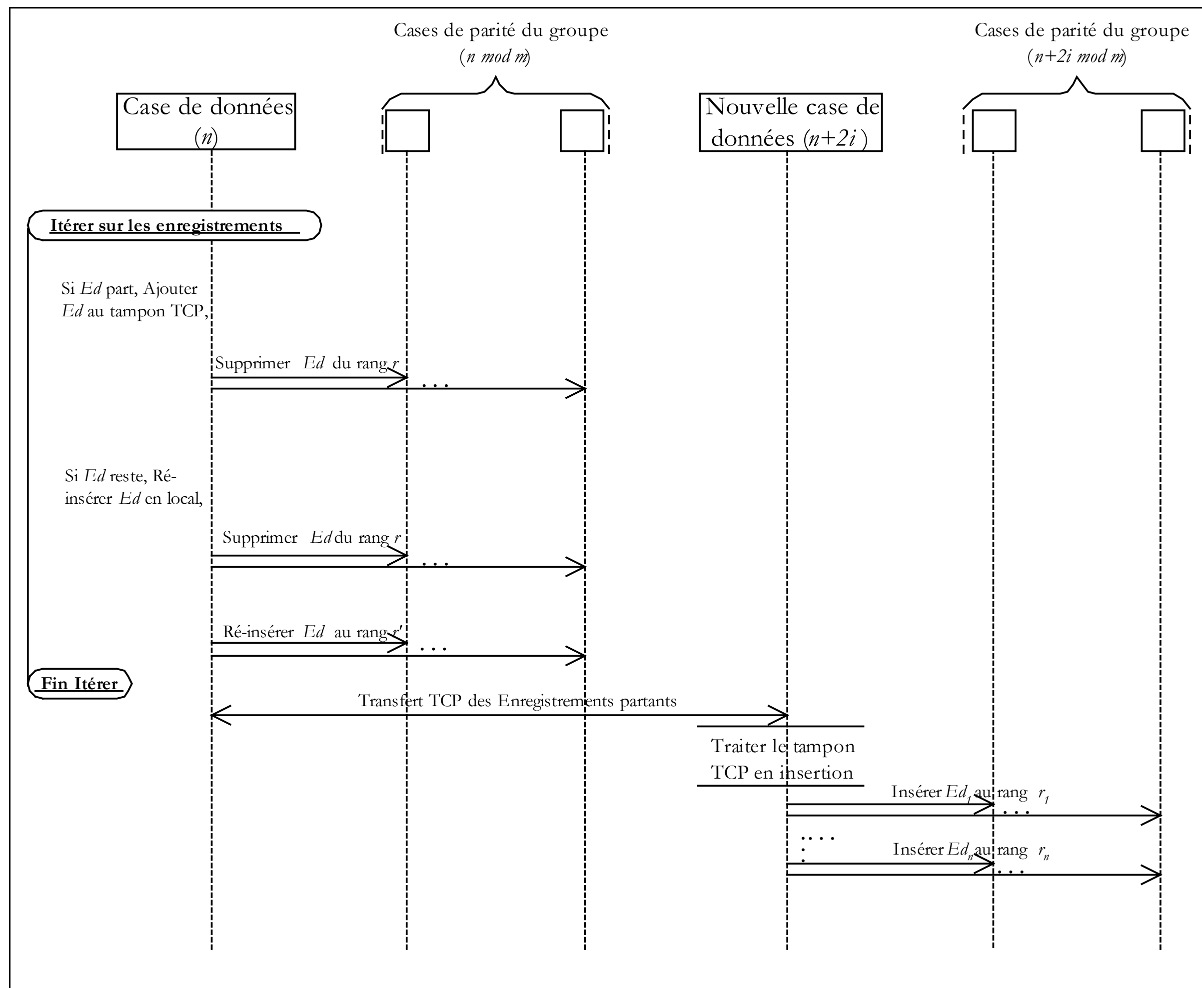


Scénario 6-4: Eclatement d'une case $LH*_{LH}$.

6.3.2 Transfert des Mises à Jour par UDP

Le Scénario 6-5 illustre l'éclatement d'une case $LH*_{RS}$ proposé par Ljungström [L00] est illustré par. Notons par *x,* désigne le nombre d'enregistrements de la case de données de numéro logique *n*. Au moment de l'éclatement, la case de données LH*, désignée par le coordinateur pour éclatement, transfère la moitié de son contenu, approximativement *x*/2 enregistrements de données, à la nouvelle case. Les enregistrements partant doivent être supprimés des cases de parité du groupe de la case de données en éclatement. Les enregistrements restants doivent être ré-insérés, puisqu'ils occupent désormais de nouveaux rangs.

A l'issue de ce scénario, chacune des cases de parité du groupe de numéro '*n* mod *m*' reçoit *x* messages de suppression et *x*/2 messages d'insertion, donc un total de 1,5*x* messages. De surcroît, chacune des cases de parité du groupe, de numéro '*n* + 2*i*', reçoit *x*/2 messages d'insertion. Notons que dans le cas particulier, où la case *n* et la nouvelle case appartiennent au même groupe, chaque case de parité recevra 2*x* messages de mises à jour.



Scénario 6-5: Eclatement d'une case de données LH*RS [L00]

Dans [L00], l'ensemble des messages de mises à jour sont acheminés à leurs destinataires moyennant le protocole UDP. Les expérimentations ont montré des incohérences des cases de parité. En effet, sachant que le protocole UDP n'est pas fiable[7], certains messages sont perdus dans le réseau, d'autres évacués de la pile UDP, et même deux messages visant le même rang, dont celui d'insertion précède celui de suppression, causent des inconsistances. Les inconsistances causées sont difficiles à détecter et corriger.

Dans [M00], nous avons proposé un scénario d'éclatement remédiant aux majeurs inconvénients dus au transfert des mises à jour par UDP aux cases de parité. Nos propositions sont décrites dans la section suivante.

6.3.3 Transfert des Mises à Jour par TCP

Dans ce qui suit, nous décrivons le nouveau mode de transfert des mises à jour vers les cases de parité et le déroulement du scénario.

[7] Transmission fiable, sans duplication de paquets et respect de l'ordre entre les messages, tel que ordre émission est égal ordre réception.

6.3.3.1 Transfert des Mises à Jour

Pour le transfert des mises à jour, le scénario d'éclatement proposé regroupe l'ensemble des messages de mises à jour dans un tampon envoyé par TCP/IP. Nous avons mis en œuvre une stratégie permettant de réduire le nombre de requêtes de mises à jour. La stratégie consiste à unifier les messages de suppression et de réinsertion, destinés aux cases de parité du groupe de la case de données qui éclate, et ce spécifiquement dans le cas où l'enregistrement de données ne migrerait pas vers la nouvelle case. En effet, dans ce cas particulier, il s'agit d'une suppression de données d'un enregistrement de données d'un rang *r* et d'insertion des même données à un rang *r'*. Le message se présenterait comme suit : « supprimer les données au rang *r* et réinsérer au rang *r'* ». Cette alternative réduirait le nombre de messages destinés aux cases de parité du groupe g ($g = n \bmod m$) de la case qui éclate, de $1,5*x$ à x messages.

6.3.3.2 Déroulement

Le Scénario 6-6 illustre l'éclatement d'une case de données $LH*_{RS}$. Dans ce qui suit, nous décrivons le déroulement du scénario au niveau de chaque entité impliquée.

<u>*Case de données n*</u>

Tout en parcourant la case $LH*_{LH}$, pour transférer les enregistrements de données partants vers la nouvelle case, le tampon de mise à jour, destiné aux cases de parité, est constitué. Ce dernier est une collection de structure,

S
- *Clé enregistrement*,
- *Attributs*,
- *Rang* : occupé par l'enregistrement, et duquel il devrait être supprimé.
- *Nouveau rang* : occupé par l'enregistrement après éclatement, et duquel il devrait être réinséré,
- *Un caractère séparateur* '|'.

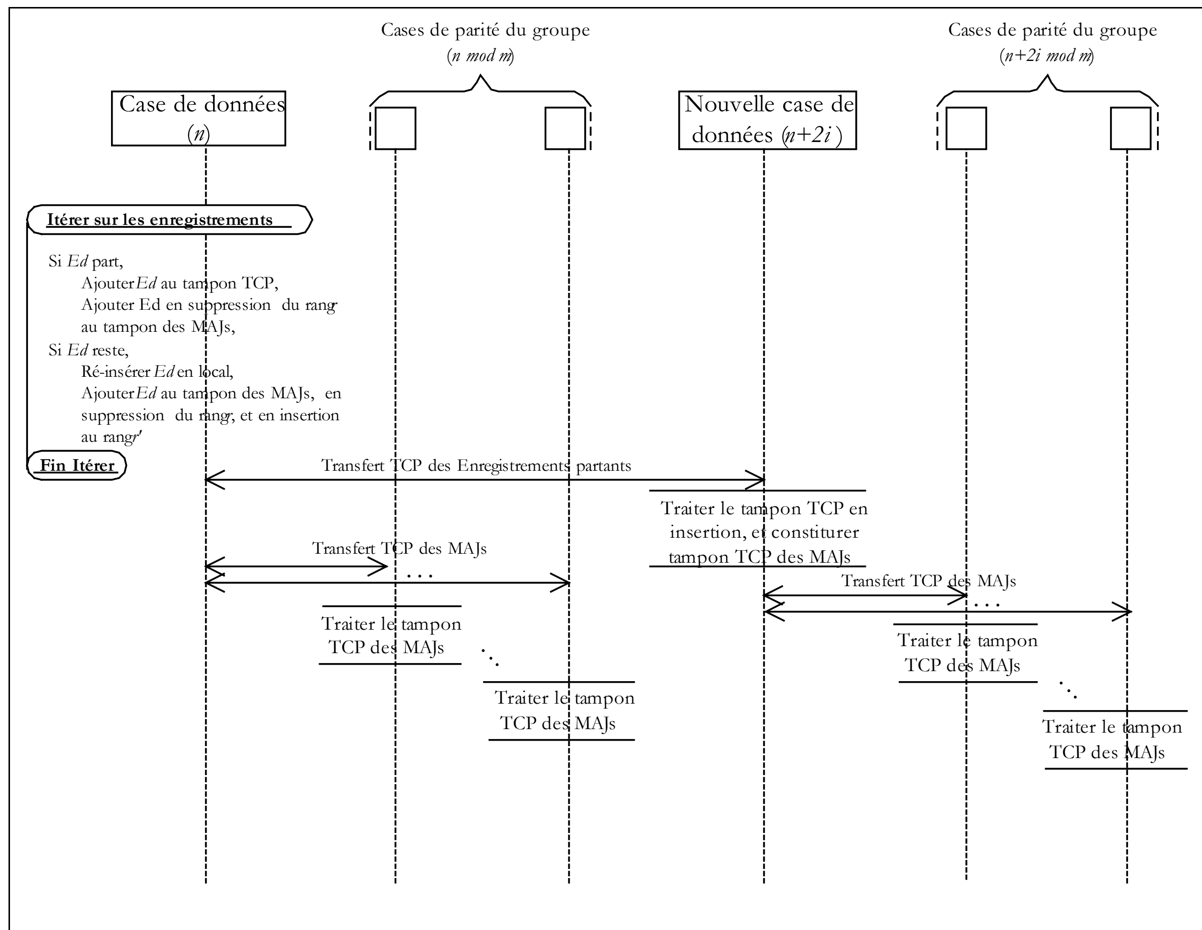
Il y'a lieu de distinguer deux cas :

1. L'enregistrement de données appartient à une *case-LH* qui migre,

   L'enregistrement de données doit être traité au niveau des cases de parité du groupe auquel la case *n* appartient, en suppression au niveau de l'enregistrement de parité de clé *Rang*. Quant à *Nouveau rang*, ce dernier est mis à –1.

2. L'enregistrement de données appartient à une *case-LH* qui reste,

   L'enregistrement de données doit être traité au niveau des cases de parité du groupe auquel la case *n* appartient, d'abord, en suppression au niveau de l'enregistrement de parité de clé *rang*, puis en réinsertion au niveau de l'enregistrement de parité de clé *Nouveau rang*.



Scénario 6-6: Eclatement d'une case de données LH*$_{RS}$ proposé.

Nouvelle case de données

En ce qui concerne, la nouvelle case de données, d'adresse logique '$n + 2^i$', celle-ci doit transmettre des mises à jour aux cases de parité de son groupe. La constitution du tampon de mise à jour TCP se fait de la même manière que la case de données *n*, et ce au fur et à mesure du traitement en insertion des enregistrements de données, provenant de la case de données *n*. Le champs *S. Rang* est mis à –1 et le champs *S. Nouveau rang* indique le rang occupé par l'enregistrement de données venant d'être inséré.

Case de parité

Il est à noter qu'une case de parité exécute les instructions de Algorithme 6-5 pour traiter chaque structure *S* du tampon TCP.

Algorithme 6-5: Procédure de Traitement d'une structure S du tampon de mises à jour.

Si (*S. Rang* ≠ –1) Alors
    Supprimer de l'enregistrement de parité de clé *Rang,*
    l'enregistrement de données *Ed* (§6.2.2)
Fin si
Si (*S. Nouveau rang* ≠ –1) Alors
    Insérer dans l'enregistrement de parité de clé *Nouveau r*
    l'enregistrement de données *Ed* (§6.2.1)
Fin si

6.3.4 Impact de l'Architecture de Cases sur le Scénario d'Eclatement

Le Scénario 6-6 s'exécute de façon différente par rapport, aux architectures décrites dans le chapitre précédent.

6.3.4.1 Architecture SDDS2000

Dans l'architecture SDDS2000, avant d'établir une connexion TCP/IP, la case émettrice et la case réceptrice se synchronisent pour établir la connexion. En effet, toute case émettrice demande une acceptation de connexion par l'envoi d'un message UDP, et attend l'acceptation de sa demande.

6.3.4.2 Architecture SDDS-TCP

Il y'a lieu de noter que le scénario d'éclatement a été littéralement changé avec l'architecture SDDS-TCP, et ce en ce qui concerne le mode d'établissement des connexions TCP/IP. En effet, dans l'architecture SDDS2000 avant d'établir une connexion TCP/IP, la case émettrice et la case réceptrice se synchronisent pour établir la connexion. Dans l'architecture SDDS-TCP, la case émettrice prépare et envoie directement le tampon à envoyer. Il est à noter que dans le scénario d'éclatement les couples de cases concernés sont [Case de données en éclatement – Nouvelle case de données], [Case de données en éclatement – chaque case de parité de son groupe] et [Nouvelle case de données – chaque case de parité de son groupe].

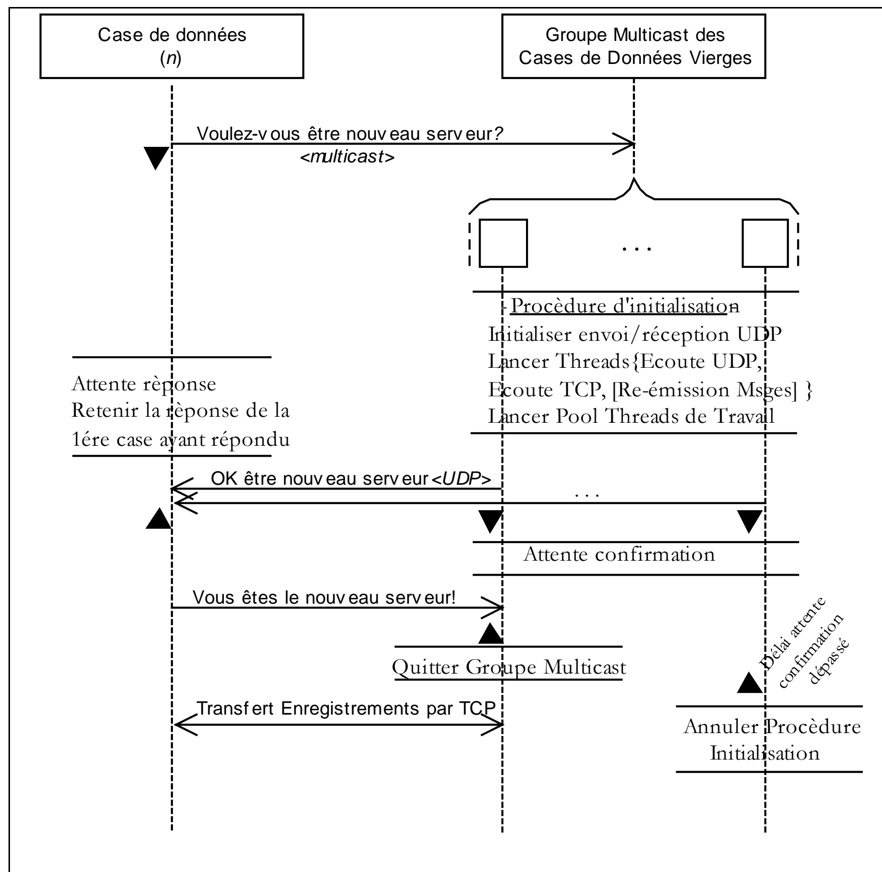
6.3.4.3 Architecture SDDS-IP@

L'impact de l'architecture SDDS-IP@, est au niveau du choix de la nouvelle case de données (Scénario 6-7), et de *k* cases de parité en cas de création d'un nouveau groupe (Scénario 6-8). La case de données en éclatement contacte les cases candidates par multicast, et en cas de réponse répond positivement à une seule en lui envoyant un message de confirmation. A la réception du message de confirmation la case de données se déconnecte du groupe multicast, et adhère aux cases du fichier.

6.3.5 Discussion de Variantes

L'éclatement réseau de LH* est coûteux, une stratégie de transfert de mises à jour vers les cases de parité plus performante est requise. En effet, dans le cas d'un fichier $LH*_{RS}$ *k*-disponible, il y'a lieu d'envoyer:

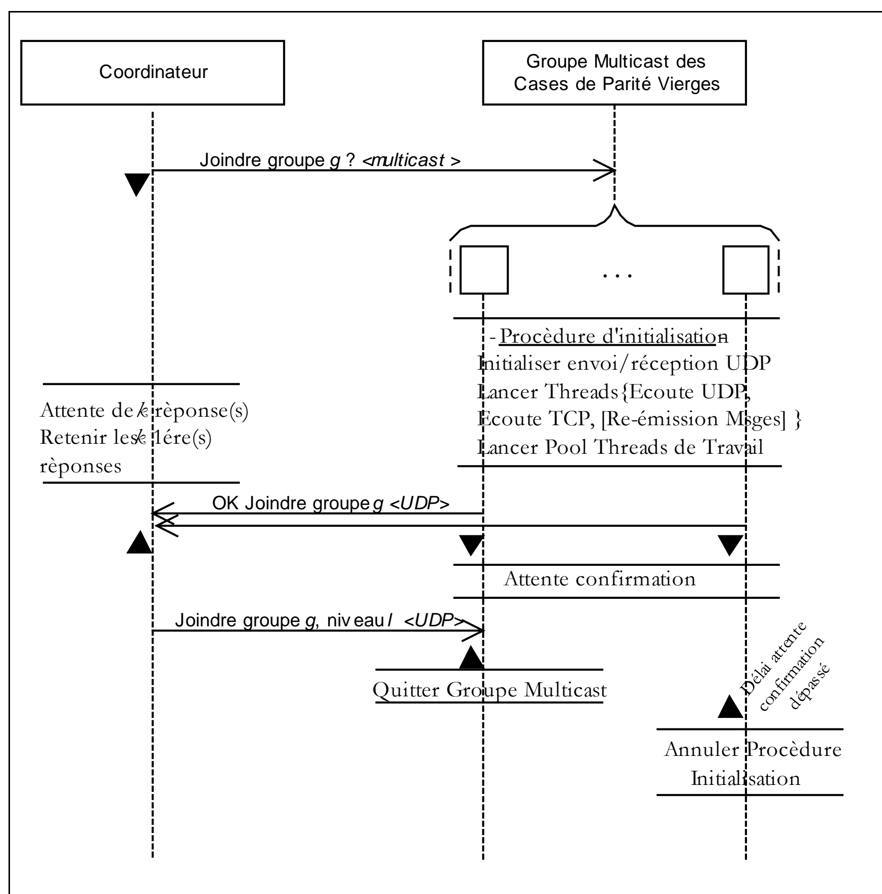
- Un tampon de la case en éclatement à la nouvelle case de taille approximative de $x/2$ enregistrements
- *k* tampons de mises à jour de la case *n* en éclatement aux cases de parité de son groupe. Tel que *k* désigne le niveau de disponibilité du groupe de la case *n*. Chaque tampon a une taille approximative de *x* enregistrements.
- *k'* tampons de mises à jour de la nouvelle case aux cases de parité de son groupe. Tel que *k'* désigne le niveau de disponibilité du groupe de la case $n+2^i$. Chaque tampon a une taille approximative de $x/2$ enregistrements.



Scénario 6-7: Ajout d'une case de données dans l'architecture SDDS-IP@.

La case de données *n* en éclatement prend en charge l'envoi de *k*+1 tampons par TCP/IP. La nouvelle case se charge de l'envoi de *k'* tampons par TCP/IP. Quand *k* augmente, l'envoi des tampons de mises à jour pénalise les cases de données, et le temps de réponse se dégrade. Pour pallier ce problème, une solution consisterait à décharger les cases de données de la propagation des tampons de mise à jour, et reléguer la tâche aux cases de parité. Les cases de données seront responsables du transfert des tampons de mise à jour juste pour une case de parité. Cette dernière soit retransmet le tampon reçu pour l'ensemble des cases de parité, ou le transmet juste pour la case de parité suivante, dite case de parité *sœur*. Dans le dernier cas, le transfert des tampons de mises à jour se fait en cascade.

Afin de paralléliser le traitement, dans le cas où les deux cases de données appartiendraient au même groupe, nous choisissons un ordre de transfert différent pour chacune des cases. La première case de parité pour la case de données en éclatement est la case de niveau 0, alors que la première case de parité pour la nouvelle case de données est celle de niveau *k*.



Scénario 6-8: Attacher *k* cases de parité à un groupe de parité dans l'architecture SDDS-IP@.

6.4 Scénario d'ajout d'une Case de Parité à un Groupe

Dans ce qui suit, nous commençons par présenter le scénario d'augmentation du niveau de disponibilité d'un groupe (illustré dans le Scénario 6-9), et consistant simplement en l'ajout d'une case de parité au groupe. Puis, nous décrivons deux scénarios implantant chacun un protocole de communication différent.

6.4.1 Création d'une Case de Parité par UDP

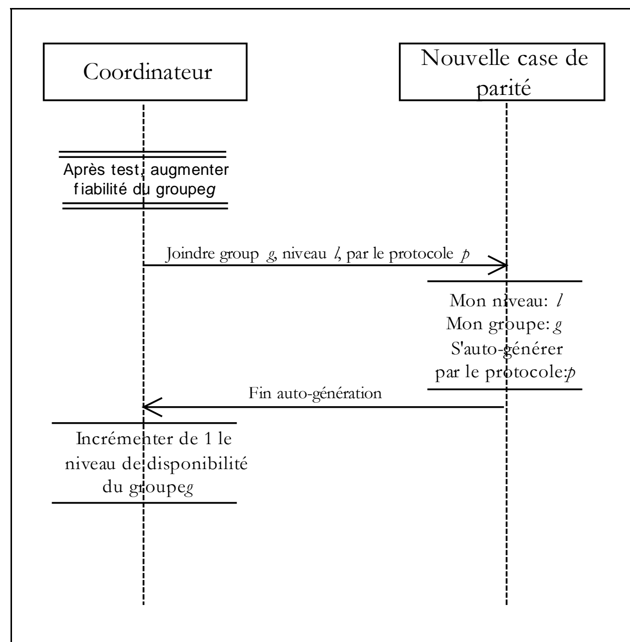
La nouvelle case de parité « s'auto-crée », en interrogeant les cases de données de son groupe. Dans ce qui suit, nous commençons par illustrer le scénario, puis discuter les différentes facettes de ce scénario (Scénario 6-10).

A un instant *t*, si un enregistrement de données de rang *r* est mis à jour, et tel que l'enregistrement de parité correspondant à ce rang n'a pas encore été calculé, cette mise à jour est inaperçue par la nouvelle case de parité. Ceci dit, le cas contraire est problématique, car il faut prévoir une procédure au niveau de chaque case de parité pour qu'elle demande les récentes manipulations ayant affecté chaque case de données. Ainsi, chaque case de données doit sauver les messages de mise à jour reçus depuis la réception du message '*Nombre d'enregistrements !*' Jusqu'au message '*Fin auto-génération !*'

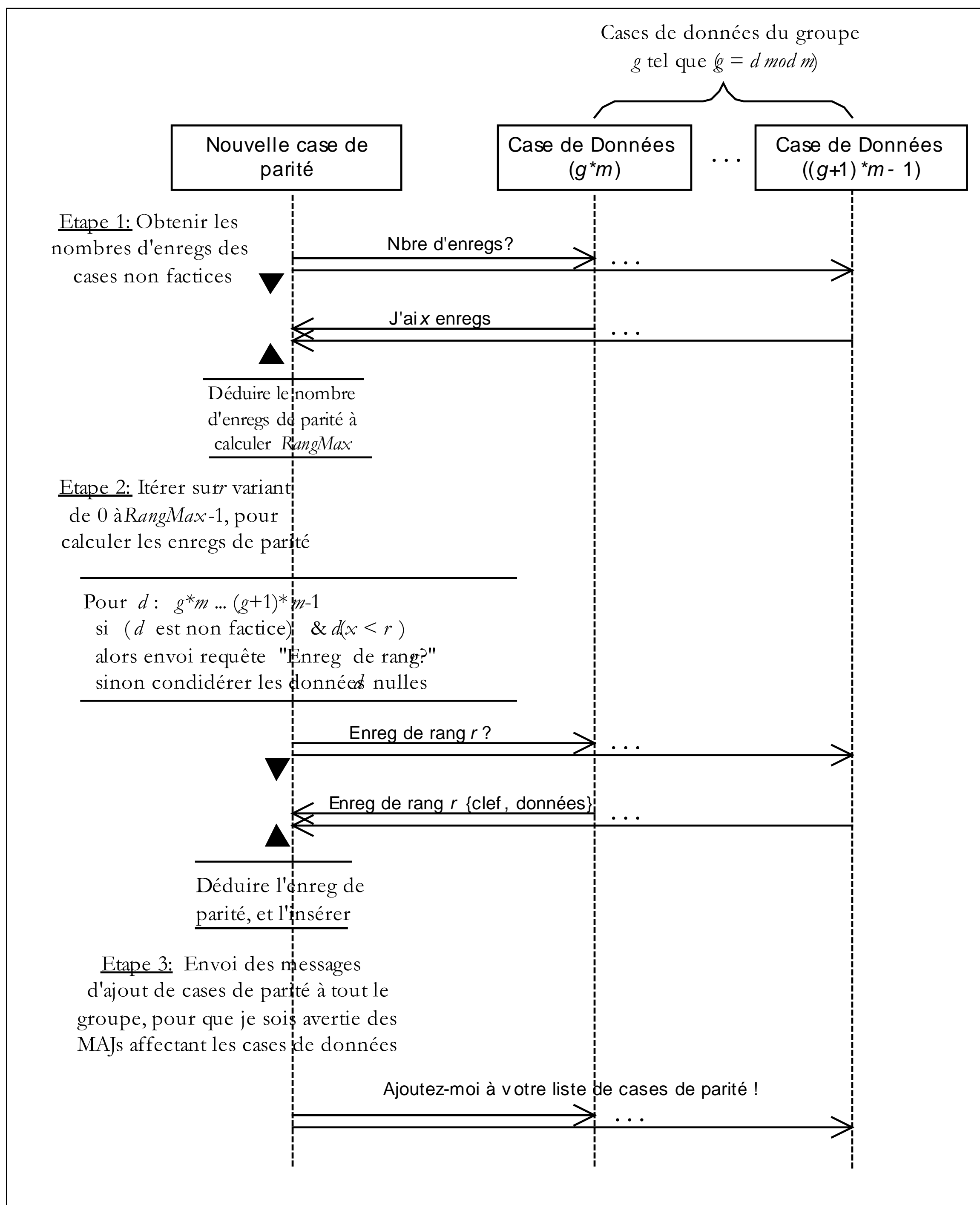
Remarquons également, si la taille du fichier augmente entre temps, c'est à dire, une nouvelle case de données est générée, et que la nouvelle case de parité appartient au dernier groupe, le coordinateur devrait soit informer la case de parité de l'ajout d'une case de donnée, qui désormais n'est plus factice. Une solution serait de retarder l'éventuel éclatement, au niveau du coordinateur, jusqu'à réception du message '*Fin auto-génération !*' ou que la nouvelle case de parité n'en tient compte qu'après avoir fini son « auto-génération ».

Notons aussi, vu que le protocole UDP n'est pas fiable, dans le cas où le message d'interrogation de la case de données '*Enregistrement de rang r ?*', ou le message de réponse '*Enregistrement de rang r {clé + données}*', destiné à la nouvelle case de parité se perd, l'enregistrement de parité en question n'est pas calculé, et il faut prévoir une procédure qui ré- interroge la case de données en question.

Dans ce qui suit, nous présentons un scénario se basant sur le protocole TCP, palliant plusieurs des majeurs inconvénients dus au protocole UDP, notamment la non-fiabilité et la restriction sur la taille des messages.



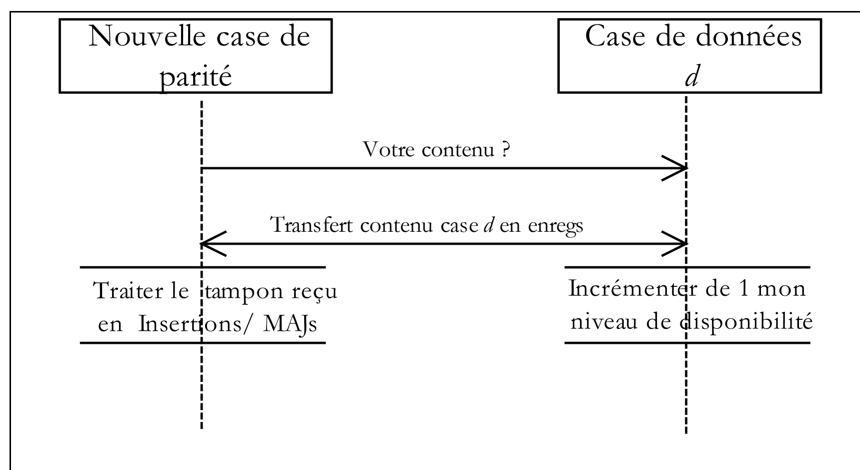
Scénario 6-9: Ajout d'une Case de Parité [Coordinateur – Case de Parité].



Scénario 6-10: Ajout d'une Case de Parité par UDP.

6.4.2 Création d'une Case de Parité par TCP/IP

L'interaction entre la nouvelle de case de parité et une case de données *d* non factice du groupe *g* se présente comme suit,



Scénario 6-11: Ajout d'une Case de Parité par TCP/IP.

Un des majeurs avantages de scénario est qu'après avoir envoyé son contenu à la nouvelle case de parité, la case de données tient compte de la présence d'une nouvelle case de parité affecté à son groupe, et pourra répercuter les manipulations client sur cette case de parité.

Par contre le désavantage, est qu'à la réception du message '*Votre contenu ?*', la case de données, et plus particulièrement un processus de travail gère la connexion TCP/IP établie, et parcourt les enregistrements de données de la case LH*, tout en remplissant le tampon avec la structure suivante :

- Clé de l'enregistrement de données,
- Champs Attributs,
- Séparateur '|'.

Notons que cette mobilisation est au détriment du temps de réponse. La question qui se pose alors quelle serait la meilleure alternative entre interroger la case de données par l'envoi de messages UDP ou mobiliser un processus de travail pendant un certain temps ?

6.4.3 Scénario Création Case de Parité vs Architecture SDDS

Le déroulement du scénario de création d'une case de parité par TCP (décrit en §6.4.2) diffère d'une architecture à une autre. Dans ce qui suit, nous décrivons l'impact de chaque composant sur le scénario.

6.4.3.1 Architecture SDDS2000

La nouvelle case de parité procède au maximum en $x \leq m$ étapes, sachant que x est le nombre de cases de données non factices du groupe g. A chaque étape, elle se connecte à une case de données, pour que cette dernière, lui envoie son contenu. A la réception du contenu d'une case de données, la case de parité traite le tampon reçu en insertion/ mise à jour respectivement selon les algorithmes respectivement décrits dans §6.2.1 et §6.2.3.

6.4.3.2 Architecture SDDS-TCP

Par rapport à SDDS2000, le scénario change vu que dans SDDS2000, plusieurs connexions TCP/IP peuvent être acceptées. Le squelette de l'algorithme exécuté par la

nouvelle case de parité allouée, est montré ci-dessous au niveau des deux *threads* impliqués : étant un *thread* de travail et le *thread* d'écoute TCP/IP.

Processus de Travail
Pour chaque case de données non factice de mon groupe:
Envoyer un message "*Envoyez-moi votre contenu!*"
Attendre jusqu'à ce que tous les tampons provenant des cases de données interrogées arrivent, et sont reçus et traités.
Processus d'écoute TCP
Attente de demandes de connexion,
Accepter les demandes de connexion dans l'entête du tampon une fois analysée, l'identificateur du traitement déterminé. S'il s'agit du contenu d'une case de données, traiter le tampon reçu en mise à jour de mon contenu,
Si tous les tampons attendus ont été reçus, signaler l'événement au thread de travail, afin de chronométrer le temps de création de la case de parité.

Des améliorations ont été apportées au scénario ci-dessus décrit. Notamment, en déchargeant le *thread* d'écoute TCP de la tâche de traitement des tampons reçus. Ainsi, chaque tampon reçu sera traité par un processus de travail.

Le parallélisme ne peut être absolu, car tous les *threads* de travail accèdent en mise à jour à la même structure de données, ce qui induit à un traitement séquentiel un par un des tampons reçus. Une stratégie plus sophistiquée consistera en le traitement en parallèle des tampons, de sorte que quand un processus finit le traitement de x enregistrements de rangs compris dans l'intervalle $[r, r+x-1]$. Il autorise un *thread* de traiter les enregistrements de mêmes rangs mais contenus dans un autre tampon, et ainsi de suite. Il est à noter que plus la taille de l'intervalle est petite, plus elle induit du parallélisme et augmente la synchronisation des *threads*.

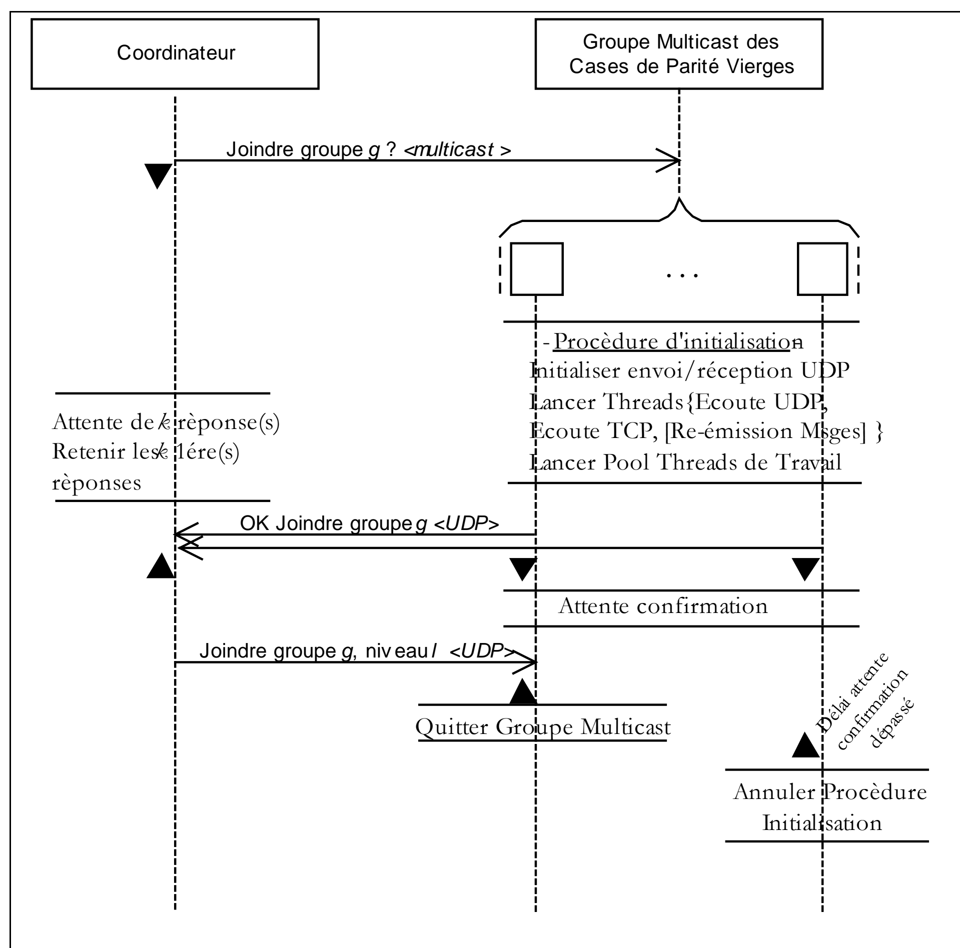
6.4.3.3 Architecture SDDS-IP@

L'architecture SDDS-IP@ modifie la procédure du choix de la nouvelle case de parité. Comme c'est illustré dans le Scénario 6-12, le coordinateur contacte les cases potentielles par multicast, et en cas de réponse répond positivement à une seule case en lui envoyant un message de confirmation. A la réception du message la case de parité se déconnecte du groupe multicast, et adhère au groupe de parité dont le niveau de disponibilité est augmenté.

6.4.4 Discussion

Si le fichier augmente de taille et qu'une case de donnée supposée factice pour la nouvelle case de parité ne l'est plus. Il est impératif d'avertir la case de parité. Nous distinguons deux solutions :

La première solution consiste à retarder l'éclatement jusqu'à réception du message « *Fin auto-génération !*». Ce dernier est envoyé par la nouvelle case de parité au coordinateur. Le coordinateur doit garder alors trace des créations de cases de parité en cours, et en cas d'éclatement mettant en jeu une case de données appartenant à un groupe, dont le niveau de disponibilité est en cours d'augmentation, l'éclatement est retardé. Notons que la case de données peut être factice si la case est sélectionnée 'nouvelle case de données' ou 'case de données non factice' si la case de données est désignée pour éclater.



Scénario 6-12: Ajout d'une case de parité dans l'architecture SDDS-IP@.

La deuxième solution consiste à ne pas retarder l'éclatement d'une case de données. Ainsi, nous sommes en face de trois cas, selon si la case était une case de données factice ou une case de données est désignée pour éclater.

1. Une case de données du groupe reçoit un ordre d'éclatement : nous distinguons deux cas selon si l'ordre d'éclatement précède ou non le message de transfert de contenu. Dans le cas où l'ordre d'éclatement précéderait le message de transfert de contenu, cela signifie que la case de données n'a pas été encore contactée par la nouvelle case de parité, au moment du transfert de son contenu, elle doit transférer son contenu après éclatement. Et donc, à la réception du message de transfert de contenu, ce dernier n'est traité qu'à la fin de la procédure d'éclatement. Par contre, dans le cas où le message de transfert de contenu précéderait l'ordre d'éclatement, c'est que la case de données a été déjà contactée par la case de parité, cette dernière recevra un tampon de mise à jour.
2. Une case de données du groupe est désormais plus factice : le coordinateur étant le contrôleur d'éclatement informera la nouvelle case de parité de l'ajout d'une case de données.

Notons que les différents cas de figures et les solutions correspondantes, développés ci-dessus sont valides au cas où plusieurs cases de données du groupe seraient impliquées dans une procédure d'éclatement.

6.5 Scénario de Récupération

Dans cette section, nous décrivons et discutons la détection de pannes, des scénarii de récupération de cases de données, cases de parité, et récupération mixte. Dans un premier temps, nous discutons la détection des échecs de sites, puis nous énonçons le corps des procédures de décodage, pour enfin montrer la procédure de récupération au niveau de la case gestionnaire de récupération.

6.5.1 Détecteur de pannes

Un détecteur de pannes ne peut pas être parfait. Il a le choix d'émettre périodiquement des messages '*Etes-vous en vie ?*' et attendre les réponses '*Je suis Vivant !*' ou être passif et attendre des alertes parvenant des entités impliquées. La première stratégie est coûteuse mais prévenante, et s'oppose à la deuxième stratégie qui est laxiste et pas coûteuse.

6.5.1.1 Stratégie *Sweep & Repair*

Cette stratégie évoquée par Weatherspoon et Kubiatowicz. [WK02], prêche la mise en œuvre d'un processus de détection d'échecs. Ce dernier vérifie périodiquement la disponibilité des cases de chaque groupe, et déclenche une procédure de récupération à la volée de cases dès détection d'échecs. Notons que, l'attente d'un événement déclencheur de récupération, de type manipulation client ou auto-détection, primo influe le temps de réponse du système et secundo peut conduire à un échec catastrophique. En effet, plusieurs nœuds peuvent tomber en panne, ce qui augmente le nombre de cases en échec, et peut rendre le nombre de cases disponibles dans un groupe inférieur à m. m étant le nombre minimum de cases disponibles pour pouvoir récupérer.

6.5.1.2 Stratégie Passive

Le coordinateur peut être averti de l'échec d'une case soit par le client ou par une des cases du fichier.

Détection d'échecs suite à une manipulation du client

- *Détection de l'échec d'une case de données* : Nous distinguons deux cas selon l'acquittement ou non des manipulations client: (1) Dans le cas où toute manipulation client serait acquittée, quelle que soit la manipulation client, insertion, suppression ou recherche d'un enregistrement, si le délai d'attente maximum est dépassé et le nombre de relances est atteint, le client adresse sa requête au coordinateur. Ce dernier en premier temps vérifie qu'il ne s'agit pas d'une erreur d'adressage. En cas d'erreur d'adressage, il informe le client de l'adresse IP de la case en question, sinon il procède au test de la disponibilité de toutes les cases de données et de parité du groupe soupçonné en échec. (2) Dans le cas où les manipulations client ne seraient pas acquittées, l'échec d'une case de données ne peut pas être perçu par le client suite à une insertion ou suppression. Par contre, le client peut signaler un échec en cas de non-réception d'une réponse à

une requête de recherche. Par conséquent, il envoie sa requête de recherche au coordinateur. Ce dernier soit informe le client de la nouvelle adresse de la case ou diagnostique le groupe en question.

- *Détection de l'échec d'une case de parité* : Nous distinguons également deux cas, suivant qu'un message de mise à jour d'une case de données aux cases de parité de son groupe, est suivi par un accusé de réception ou non. Dans le cas où, la case de parité ne répondrait pas, après épuisement du nombre de relances, le coordinateur est averti. Ce dernier, vérifie qu'il ne s'agit pas d'une erreur d'adressage et dans ce cas il informe la case de données de la nouvelle adresse de la case de parité. Dans le cas contraire, il procède au test de disponibilité des cases de données et de parité du groupe en question. Notons qu'en absence d'accusés de réception, un échec d'une case de parité ne peut être détecté, que suite à au test de détection effectué par le coordinateur.

Cas d'auto détection d'échecs

Le cas d'auto détection exclut tout avertissement de la part d'un client. Cette stratégie escompte une détection d'échecs lors des scénarii suivants :

- *Scénario d'éclatement* : Lors du scénario d'éclatement, il se pourrait que la case de données *n* soit en échec et/ou des cases de parité soit du groupe de la case de données en éclatement ou du groupe de la nouvelle case de données soient en échec. Dans le scénario d'éclatement, le coordinateur devrait s'assurer que la case de données *n* a bien reçu l'ordre d'éclatement, et l'a exécuté, idem les cases de données : celle en éclatement et la nouvelle doivent réussir à acheminer les mises à jour vers les cases de parité de leurs groupes respectifs. Dans le cas contraire, les cases de parité en échec sont récupérées à partir du nouveau contenu des cases de données, et donc aucun besoin de ré-itérer la transmission des tampons de mises à jour.
- *Scénario de haute disponibilité* : Lors du scénario de haute disponibilité, à l'appel de la nouvelle case, il se pourrait qu'une case de données n'envoie pas son contenu. Il faudrait dans ce cas récupérer le(s) case(s) de données en échec, puis avertir la case de parité en auto-génération d'actualiser son contenu par rapport aux cases récupérées.

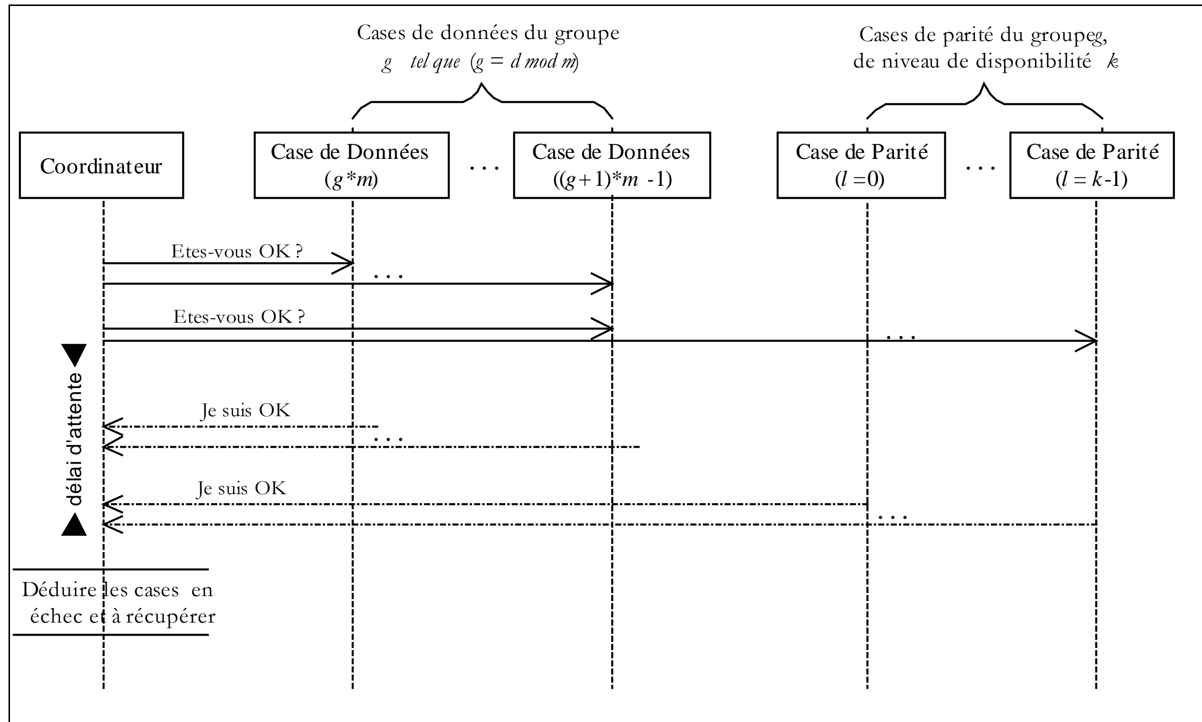
Vu la complexité de ces scénarios, les différents contrôles à implanter et sachant que le prototype a pour but d'évaluer la complexité de la récupération de cases, dans la mesure où c'est possible, nous simplifions le scénario de récupération et nous nous limitons à l'énumération des cas d'échec. En première étape, nous généralisons le traitement des échecs, qui aboutit dans tous les cas à notifier le coordinateur. Ce dernier diagnostique le groupe en question; par la suite nous distinguons plusieurs cas dépendant de qui va prendre en charge la procédure de récupération.

Les sous-sections suivantes exposent la procédure de récupération au niveau, du coordinateur et de la case gestionnaire de récupération.

6.5.2 Diagnostic d'un Groupe de Parité

Une fois, notifié d'un échec d'une ou plusieurs cases appartenant au groupe *g*, le coordinateur vérifie la disponibilité des cases de données et des cases de parité du groupe *g*; et conclut par corriger l'erreur d'adressage ou en cas d'échec et possibilité de récupération, désigner un gestionnaire de récupération.

Notons que le coordinateur a le choix soit de faire un diagnostic de tout le groupe afin de récupérer toutes les cases en échec ou de se contenter de la réponse de *m* cases pour entamer la récupération des cases soupçonnées en échec. Dans le dernier cas, des cases du groupe pourraient être en échec et ne sont pourtant pas récupères. Le Scénario 6-13 montre le premier cas.



Scénario 6-13: Diagnostic d'un groupe de parité.

Désignons par,

nd : le nombre de cases de données indisponibles,
np : le nombre de cases de parité indisponibles,
k : niveau de disponibilité du groupe *g*, (càd le nombre de cases de parité).

Dans une première étape, le coordinateur teste la possibilité de récupérer. Ainsi, nous distinguons trois cas :

Algorithme 6-6: Test de Récupération.

Si ($nd = 0$ & $np = 0$) *Alors*
Il s'agit d'une fausse alerte, soit la case était surchargée, ou
un échec du lien de communication entre les deux entités.
Sinon Si ($k - nd - np < 0$) *Alors* le groupe est irrécupérable.
Sinon Récupération Possible
Fin Si

Dans le cas de récupération possible, le coordinateur désigne un gestionnaire de récupération, et ce suivant le nombre de cases de parité disponibles. En effet, si toutes les cases de parité sont indisponibles : $np = k$, le gestionnaire de récupération serait une case de donnée, autrement une case de parité.

Enfin, le coordinateur envoie des messages d'initialisation aux cases remplaçant, celles en échec. Pour la case de parité, le message lui communique son niveau dans la matrice génératrice G et son groupe, étant g.

6.5.3 Procédures de Décodage

Dans ce qui suit, nous commençons par expliciter les corps de la procédure de récupération de *nd* enregistrements de données d'un rang *r*, et celle de *np* enregistrements de parité de clé *r*, pour enfin présenter les deux scénarios, correspondant respectivement aux cas d'une récupération gérée par une case de parité et une par une case de donnée.

6.5.3.1 Récupération d'Enregistrements de Données

Nous distinguons deux cas de figures. En effet, notre matrice génératrice, notamment la première colonne de '1's, permet d'exécuter un décodage XOR.

(1) Cas de décodage RS

Cette procédure, sachant m enregistrements de données ou de parité soient-ils, récupère *nd* enregistrements de données, de même rang.

Algorithme 6-7: Décodage RS [L00]

*H (m*m)* : fusion de m colonnes de G, correspondant aux cases disponibles,
Inverser H.
*E (m*t)* : structure contenant m enregistrements disponibles,
Itérer sur j variant de 0..t
$b \leftarrow E^j$
Pour i de 0 à m
$R[i][j] \leftarrow XOR_{s=0}^{m-1} \; b[s] * H^{-1}[s]$
Fin Itérer

Cette procédure proposée par M. Ljungtröm [L00] recalcule les m enregistrements de données, y compris les enregistrements disponibles. Une optimisation de ce calcul serait de ne pas calculer tous les symboles, et donc nous proposons de remplacer les instructions encadrées en pointillé par les suivantes,

Algorithme 6-8: Décodage RS optimisée [M00].

*H (m*m)* : fusion de m colonnes de G, correspondant aux cases disponibles,
Inverser H.
*E (m*t)* : structure contenant m enregistrements disponibles,
Itérer sur j variant de 0..t
$b \leftarrow E^j$
Pour i de 0 à *nd*
$R[i][j] = XOR_{s=0}^{m-1} \; b[s] * H^{-1}[s][Indices[i]]$
Fin Itérer
//La structure *Indices* dénote les niveaux des cases de données en échec.

Ainsi, à chaque itération, nous économiserons le calcul de *(m – nd)* symboles, coûtant *(m - nd) * (m-1)* XORs et *(m - nd) * m* multiplications GF.

(2) Cas Particulier de décodage XOR

Le décodage XOR s'effectue dans un groupe où seulement une seule case de données *d* est en échec, et la première case de parité est disponible. Cette dernière recalcule les enregistrements de la case *d*, comme suit:

Algorithme 6-9: Décodage XOR.

*E (m*t)* : structure contenant *m* enregistrements disponibles,
Itérer sur *j* variant de 0 .. *t*
$enreg[j] \leftarrow XOR_{s=0}^{m-1} \; E[j][s]$
Fin Itérer

Remarquons que dans ce cas particulier, la récupération d'un enregistrement de données coûte *t*m* XORs.

6.5.3.2 Récupération d'Enregistrements de Parité

Cette procédure, sachant les *m* enregistrements de données, récupère *np* enregistrements de parité manquants de même rang. L'opération de décodage n'est en fait que du codage.

Algorithme 6-10: Récupération RS d'enregistrements de parité.

*D(m*t)* : structure contenant *m* enregistrements de données
Liste des clés : contient les *m* clés des enregistrements de données
Indices : tableau des niveaux des cases de parité en échec.
Itérer sur *j* variant de 0..*t*
$a \leftarrow Dj$
Pour *i* de 0 à *np*
$P[i][j] = XOR_{s=0}^{m-1} \; a[s] * G[s][Indices[i]]$
Fin Itérer

Chaque ligne de la structure *P*, constitue un enregistrement de parité, à envoyer à la case de parité de niveau $Indices[i]_{\, i\,:\,0..np}$ correspondant.

Il est à noter que dans le cas particulier où *np* est égal à 1, et qu'il s'agit de la première case de parité à récupérer, grâce à notre matrice génératrice, nous récupérons la case de parité en exécutant un calcul XOR.

Algorithme 6-11: Récupération XOR d'enregistrements de parité.

*D(m*t)* : structure contenant *m* enregistrements de données
Liste des clés : contient les *m* clés des enregistrements de données
Itérer sur *j* variant de 0..*t*
$a \leftarrow Dj$
Pour *i* de 0 à *np*
$P[i][j] = XOR_{s=0}^{m-1} \; a[s] * G[s][Indices[i]]$
Fin Itérer

6.5.3.3 Améliorations

Nous avons tenté d'améliorer le scénario en prévoyant un algorithme de traitement de chaque enregistrement ou tampon d'enregistrements à leurs réceptions (resp. pour la récupération par UDP et la récupération par TCP). En effet, dans les scénarii décrits ci-dessus, la procédure de décodage n'est enclenchée qu'une fois tous les tampons sont reçus. La procédure suivante permet de traiter chaque enregistrement. Ainsi, les tampons de récupération sont mis à jour au fur et à mesure, mais ne sont prêts à l'envoi qu'en cas de réception de toutes les réponses attendues.

Notons qu'en première étape la case de parité initialise la structure de récupération au contenu de son enregistrement de parité pour un décodage XOR et au contenu traité en utilisant l'Algorithme 6-12 pour un décodage XOR. Puis traite chaque réponse de la part d'une case disponible en appliquant l'Algorithme 6-12 ou l'Algorithme 6-13 respectivement selon qu'il s'agit d'un décodage RS ou XOR.

Algorithme 6-12: Décodage RS en ligne.

*H (m*m)* : fusion de *m* colonnes de *G*, correspondant aux cases disponibles,
Inverser *H,*
enreg.: enregistrement parvenant d'une case *b* disponible,
pos : la case *b* a la position *pos* dans la liste des cases invoquées,
Itérer sur *j* variant de 0..*t*
Pour *i* de 0 à *nd*

$$R[i][j] \wedge = Enreg[j] * H^{-1}[pos][Indices[i]]$$

Fin Itérer
//La structure *Indices* dénote les niveaux des cases de données en échec.

Le décodage XOR n'est utilisé qu'en cas de récupération d'une seule case ou un seul enregistrement de données, et est conditionnée par la disponibilité de la première case de parité du groupe en question.

Algorithme 6-13: Décodage XOR en ligne.

enreg : enregistrement parvenant d'une case *b* disponible,
récup : enregistrement à récupérer,
Itérer sur *j* variant de 0..*t*

$$récup[j] \wedge = Enreg[j]$$

Fin Itérer

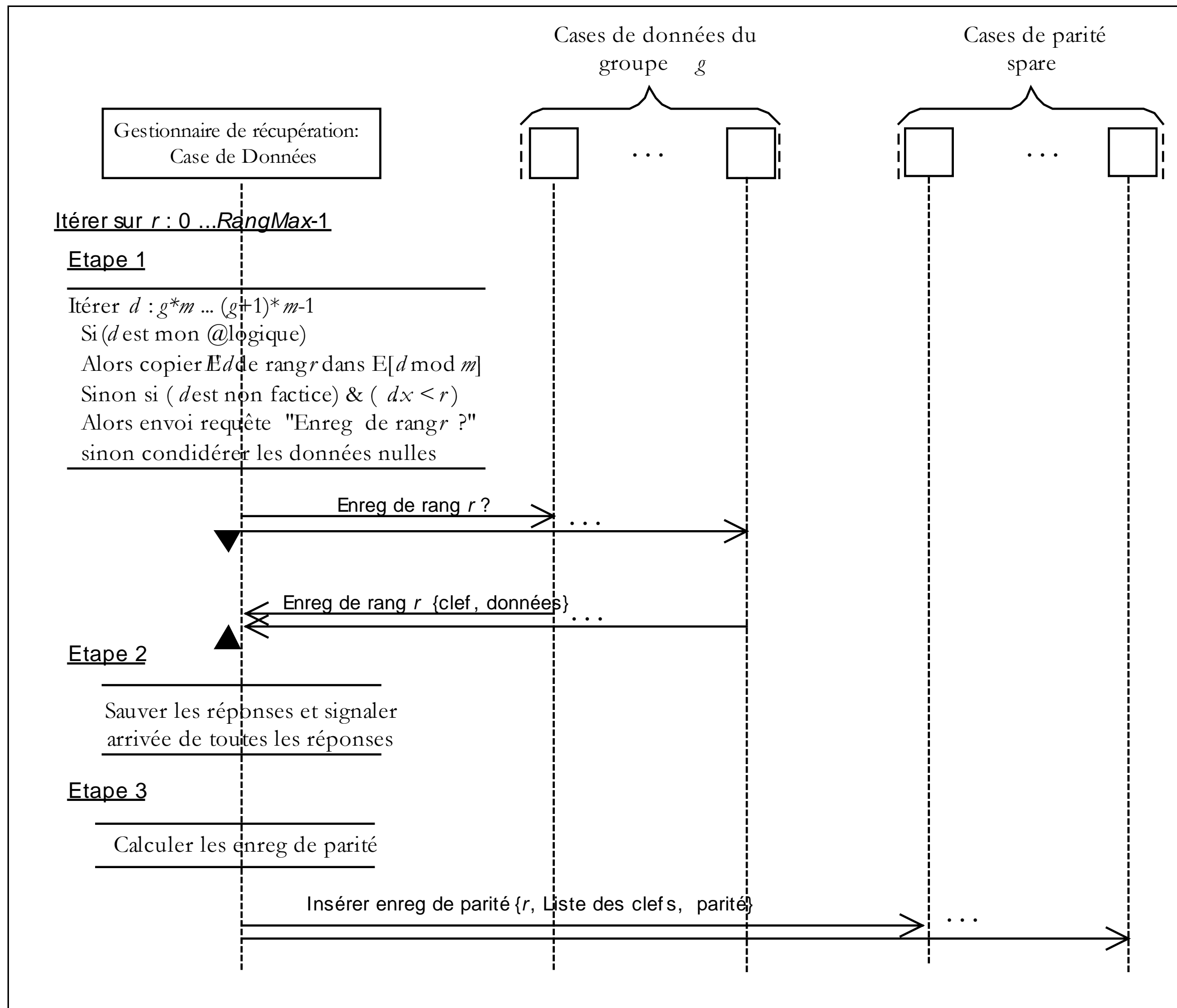
6.5.4 Récupération par UDP

La récupération par UDP de l'ensemble des cases de parité du groupe est illustrée dans le Scénario 6-14. Le Scénario 6-15 illustre le traitement d'échecs soit de cases de parité et de cases de données ou seulement de cases de données.

6.5.5 Récupération par TCP

Les scénarios de récupération se basant sur le protocole UDP, interrogent chaque case participante pour un enregistrement afin de récupérer *f* enregistrements. L'idée d'une approche basée sur TCP/IP est la suivante, le gestionnaire de récupération

interrogerait chaque case participant tranche enregistrement afin de récupérer *f**tranche* enregistrements. Le paramètre *tranche* variera en fonction des capacités de gestion de tampons du gestionnaire de récupération.



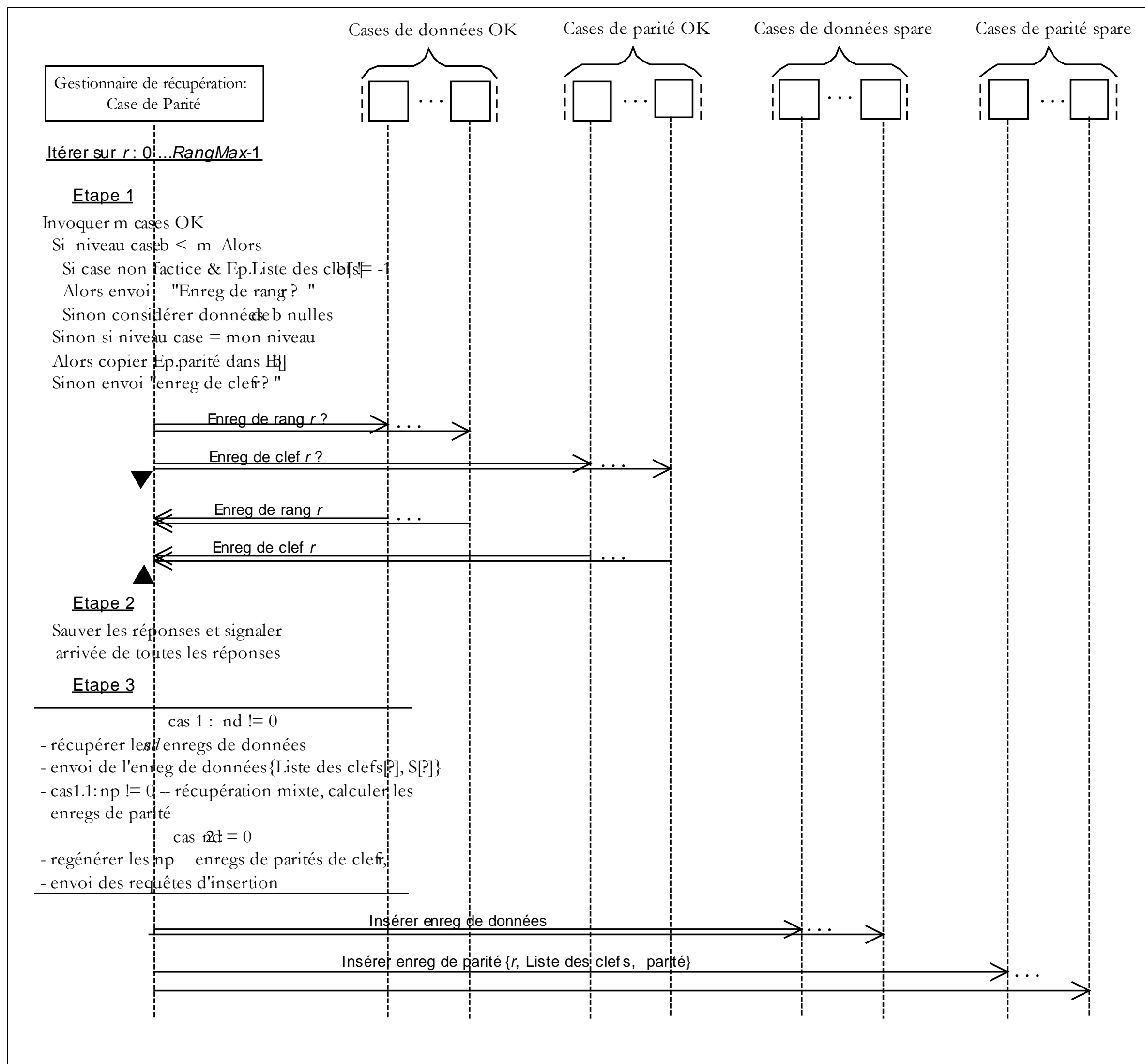
Scénario 6-14: Récupération de l'ensemble des cases de parité par UDP.

Le scénario de récupération basé sur le protocole TCP/IP change selon l'architecture des cases. Dans ce qui suit, nous décrivons le scénario dans chaque architecture de case investiguée.

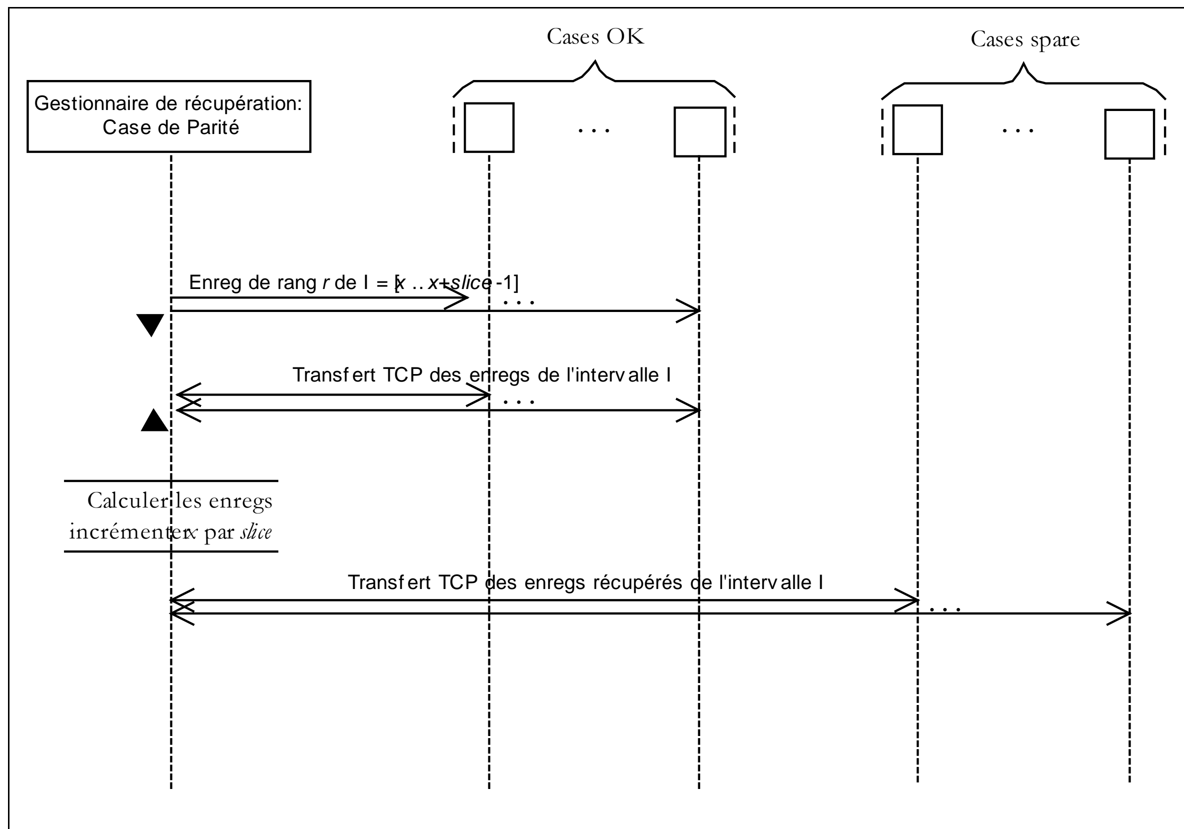
6.5.5.1 Architecture du prototype SDDS-2000

A chaque itération, le gestionnaire de récupération demande aux cases sélectionnées pour participer à la récupération, une à une, d'envoyer une tranche d'enregistrements de données correspondant aux rangs: *r*, …, *r* + *tranche* – 1. le compteur *r* est incrémenté de tranche à la fin de chaque itération. Les cases répondent dans la limite de leurs nombres d'enregistrements, et envoient au gestionnaire de récupération les enregistrements appartenant à l'intervalle [*r*, …, *r* + *tranche* – 1]. Après réception de tous les tampons attendus, les enregistrements ayant le même rang sont lus à partir des tampons reçus, et de la structure de données locale du gestionnaire de récupération, pour calculer les enregistrements des cases en échec. Les enregistrements manquants sont récupérés et sauvés dans des tampons de récupération. Enfin, le gestionnaire de

récupération envoie à chaque case de secours la tranche récupérée. Tous les tampons sont envoyés par TCP/IP.
A la fin du processus de récupération, le gestionnaire notifie le coordinateur de la fin de la récupération. Le coordinateur effectuera le changement d'adresse des cases récupérées, et notifie le changement d'adresses de cases à toute entité concernée.
Notons qu'à chaque itération, avant l'envoi aux cases de secours des tampons récupérés, les messages d'interrogation des cases disponibles sont envoyés, afin de paralléliser la réception et l'envoi.



Scénario 6-15: Récupération mixte par UDP.



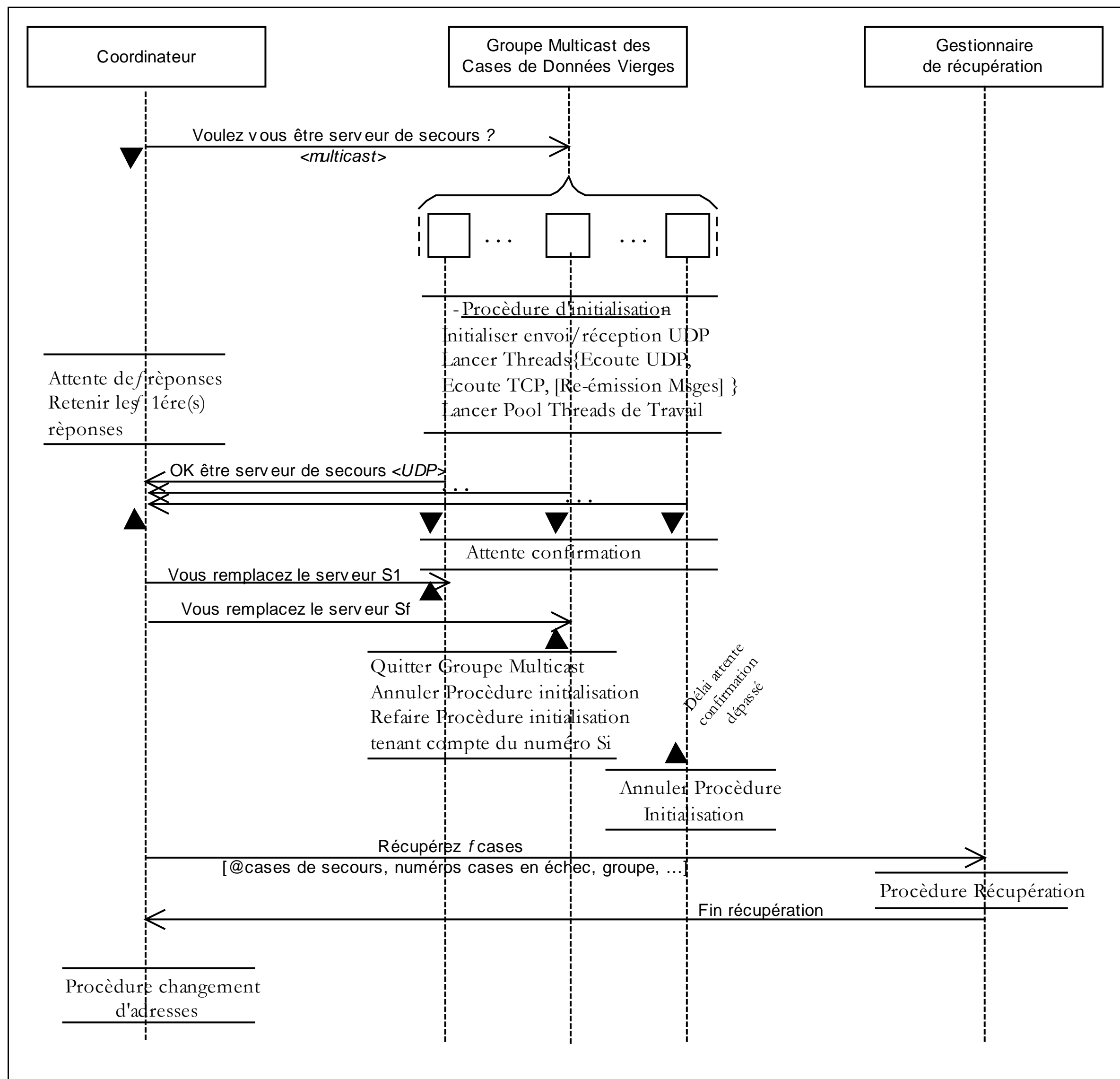
Scénario 6-16: Récupération par TCP/IP et par Tranche d'enregistrements.

6.5.5.2 Architecture SDDS-TCP

Par rapport à l'architecture SDDS2000, ce qui change c'est la gestion des connexions TCP/IP. Ces dernières ne sont plus établies par ordre séquentiel avec chacune des cases participant à la récupération, ou case de secours. En effet, le gestionnaire de récupération est préparée à accepter plusieurs connexions par TCP/IP. De la même manière, aucun traitement pré-connexion n'est fait par le gestionnaire pour envoyer les tampons de récupération aux cases de secours.

6.5.5.3 Architecture SDDS-IP@

L'impact de l'architecture SDDS-IP@, est au niveau du choix des *f* cases de secours. Comme le montre le Scénario 6-17, le coordinateur s'adresse au groupe multicast de cases de données vierges et au groupe multicast des cases de parité vierges. Le coordinateur répond positivement à *f* cases en leurs envoyant un message de confirmation. A la réception du message de confirmation une case se déconnecte du groupe multicast. En cas de manque de réponses, le coordinateur réitère sa demande.



Scénario 6-17: Choix des cases de secours dans l'architecture SDDS-IP@.

6.5.5.4 Analyse du Scénario

Afin de récupérer *f* cases de données en échec, le gestionnaire de récupération reçoit au maximum *m-1* tampons par itération de cases disponibles (en effet moins que *m-1* tampons dans le cas où certaines cases sont factices).

La récupération d'une case de données supplémentaire a un coût de décodage et un coût de communication. Soient :

> *cc :* le coût en communication réseau pour la récupération d'une case de données,
>
> *cd:* le coût en décodage pour la récupération d'une case de données,

Le coût de décodage de *f* cases de données doit être égal à *f*cd*. Bien évidemment, cette équation n'est plus valide si nous utilisons différents algorithmes de décodage, notamment le cas de récupération d'une case de données à travers le décodage XOR et le cas récupération de deux cases de données à travers le décodage RS.

Le coût inféré à la communication réseau pour la récupération de *f* cases de données doit être strictement inférieur à $f*cc$. En effet, le temps de communication nécessaire pour la réception de tampons de la part des cases disponibles est le même pour la récupération d'une case de données ou de *f* cases de données.

Soit *f* le nombre de cases en échec, #*b* le nombre d'enregistrements dans une case de parité. Le gestionnaire de récupération exécute #*Iter* égal à #*b*/ *tranche* itérations pour récupérer les cases en échec. A maximum, le gestionnaire reçoit #*Iter**(*m*-1) tampons et envoie #*Iter***f* tampons pour les cases de secours. Le nombre de tampons reçus ou envoyés dépend du nombre de cases factices et des enregistrements de données factices (identifiées si le champ liste est égal à -1). Notons que dans le dernier cas, la différence en nombre d'enregistrements doit être égale à la valeur d'une *tranche* d'enregistrements pour gagner une connexion pour réception ou envoi.

Variation de la taille de *tranche*, par conséquent du nombre d'itérations #*Iter*

$$\#Iter1*(m+f-1) - \#Iter2*(m+f-1) = (\#Iter1 - \#Iter2)*(m+f-1)$$

Variation du nombre de cases en échec *f*

$$\#Iter *(m-1) + \#Iter *f1 - \#Iter*(m-1) + \#Iter*f2 = \#Iter*(f1- f2)$$

6.6 Conclusion

Tout au long de ce chapitre, nous avons étudié des scénarii pour l'éclatement réseau d'une case de données $LH*_{RS}$, la création d'une case de parité afin d'augmenter la disponibilité d'un groupe, et des scénarii de récupération. Par rapport à chaque architecture, un scénario évolue. Dans certains cas, nous démontrons l'apport d'une amélioration théoriquement par comparaison de la complexité de chaque scénario. Nous ne pouvons quantifier l'amélioration que par expérimentation. Dans le chapitre suivant, nous rapportons les mesures de performances de chaque scénario, afin de valider nos choix architecturaux et conceptuels.

Vu que nous sommes dans un milieu réparti, nos scénarii restent valides en cas de changement de l'algorithme de codage ou de décodage, et a priori de l'algorithme de distribution des données.

CHAPITRE

7 MESURES DE PERFORMANCES DE LH*$_{RS}$

7.1 Introduction

Dans le présent chapitre, nous rapportons les résultats des expérimentations conduites. Les résultats feront l'objet d'analyses, suite auxquelles des conclusions seront tirées. Les expérimentations ont pour but d'évaluer nos choix architecturaux et de mesurer les performances des scénarios proposés. Dans une première section nous décrivons notre environnement expérimental, puis nous évaluons les optimisations du codage et décodage investiguées. Dans les sections suivantes, nous rapportons et discutons les performances des scénarios suivants :

* Création d'un fichier LH*$_{RS}$.
* Recherche d'enregistrements de données.
* Récupération d'enregistrements de données.
* Mises à jour d'enregistrements de données.
* Création de cases de parité.
* Récupération de cases de données.

7.2 Environnement Expérimental

7.2.1 Configuration Matérielle

Les expérimentations ont été conduites sur deux configurations matérielles différentes. Ce qui nous permet d'extrapoler les résultats obtenus dans d'autres configurations matérielles et de mesurer l'impact du progrès technologique touchant la vitesse des processeurs et la bande passante du réseau sur les performances de notre prototype.

Première Configuration

L'environnement expérimental de la première configuration, se compose de 6 machines Pentium 733 Mhz, chacune de 128 Mo de RAM, et exécutant Windows 2000. Les machines sont reliées par un réseau Ethernet 100 Mbps.

Deuxième Configuration

L'environnement expérimental de la deuxième configuration se compose de 5 machines Pentium 1,8 Ghz, chacune de 512 Mo de RAM, et exécutant Windows 2000. Les machines sont reliées par un réseau Ethernet 1Gbps.

Remarquons que :

(1) La vitesse des processeurs des machines de la deuxième configuration est 2.45 fois plus rapide que celles des processeurs des machines de la première configuration. Ceci induit un gain idéal des performances de traitement local de 60%.

(2) La bande passante du réseau Ethernet de la deuxième configuration est un ordre de magnitude supérieure à celle de la première configuration. Ceci induit une amélioration idéale des performances de communication de 90%.

(3) L'amélioration globale idéale est égale à :

$$\frac{60\%\ Temps\ Trait.1ére\ Config. + 90\%\ Temps\ Com.1ére\ Config.}{Temps\ Total\ 1ére\ Config.}.$$

(4) Le rapport des capacités des mémoires vives respectives à la deuxième et première configuration est égal à 2.

7.2.2 Paramètres de LH*$_{RS}$

Tout au long du chapitre, nous adoptons la terminologie suivante :

Symbole	*Description*	*Valeurs types*
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.	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
CG	Corps de Galois	CG(2^8) ; CG(2^{16})
T	Taille des attributs non-clé d'un enregistrement, exprimée en nombre de symboles	100 octets 50 symboles du CG(2^{16}) ou 100 symboles du CG(2^8).

Les expérimentations faites par M. Ljungstrom [L00] ont montré que le calcul de parité dans le corps de Galois : CG(2^8) est plus performant que dans le corps de Galois : CG(2^4). C'est ainsi que notre choix en corps de Galois s'est restreint à : CG(2^8) et : CG(2^{16}).

Nous comparons les résultats obtenus pour chacune des configurations, en prenant en considération, primo, la composante temps de traitement dépendant essentiellement de la vitesse du processeur et de la mémoire vive, et secundo, la composante temps de communication dépendant de la bande passante du réseau. En ce qui concerne les temps de communication, ces derniers dépendent de deux facteurs que sont *(i)* le *débit*, étant la quantité d'information transmise par unité de temps et *(ii)* la *latence*, étant le délai d'accès au site distant.

7.3 Optimisation du Calcul de Parité

Après avoir mené des expérimentations, il s'est avéré que le pré-calcul du *log*, (décrit en §4.3.3.2) et la nouvelle matrice génératrice (décrite en §4.3.3.1) réduisent le temps de calcul de parité.

7.3.1 Pré-calcul du *log*

Le pré-calcul du log optimise la procédure de multiplication de deux éléments d'un corps de Galois, tel que le premier élément est un coefficient de la matrice génératrice et le deuxième est un symbole. Cette optimisation est conséquente sur les processus de codage et décodage. Afin de prouver l'efficacité du pré-calcul du log, nous avons mené une série d'expérimentations consistant en la création d'une case de parité de 31250 enregistrements, et utilisant un codage Reed-Solomon dans CG(2^8). La Table 7-1 rapporte les temps de traitement en codage RS sans le pré-calcul du *log* et en codage RS avec le pré-calcul du *log* ; et ce dans chacun des corps de Galois CG(2^8) et CG(2^{16}).

	Simple Codage RS	*Codage RS + Pré-calcul Log*	*Amélioration*
CG(2^8)	1797	1728	4%
CG(2^{16})	1415	1361	3,82%

Table 7-1: Comparaison des temps de traitement de création d'une case de parité (en ms) dans la 2éme config.

Les améliorations obtenues de l'ordre de 4% valorisent la stratégie de pré-calcul du *log*, pour ce l'option est toujours activée dans les futures expérimentations.

7.3.2 Nouvelle Matrice Génératrice

Au vu de la matrice optimisée, une case de parité traitant une requête de mise à jour ou un tampon de mises à jour provenant de la première case de données d'un groupe, exécute des opérations de '*ou exclusif*' (XOR), et non des opérations de multiplications par '1' dans un corps de Galois. Afin de valoriser ce gain en calcul de parité, nous avons mené une série d'expérimentations consistant en la création d'une case de parité, et utilisant des méthodes de calcul de parité différentes. La première méthode crée une case de parité en utilisant la matrice génératrice non-optimisée, et

donc l'opération principale de codage est la multiplication dans un corps de Galois. La deuxième méthode tient compte de la ligne de '1's, donc réduit les opérations de multiplication en simples ou-exclusifs et ce chaque fois qu'une mise à jour provient de la première case de données du groupe. Enfin, par rapport à la deuxième méthode, elle intègre le pré-calcul du *log*, et est désignée dans la suite du document *Codage* RS^{++}.

La Table 7-2 rapporte les temps de traitement de création d'une case de parité par les trois méthodes dans la deuxième configuration.

	Simple Codage RS	*Codage RS + Ligne de '1's*	*Codage* RS^{++}	*Amélioration RS+Ligne de '1's / RS*	*Amélioration* RS^{++}/RS
CG(2^8)	1797	1637	1618	9%	10%
CG(2^{16})	1415	1354	1336	4,31%	5,58%

Table 7-2: Comparaison des temps de traitement de création d'une case de parité (en ms), dans la 2éme config.

Notons que,

* Les améliorations obtenues pour le corps CG(2^{16}) en moyenne de 4,31% sont moins importantes que celles obtenues dans CG(2^8) en moyenne 9%. Ceci montre que le '*ou- exclusif*' est plus efficace dans le corps CG(2^8), donc plus opérationnel sur les octets.

* Les améliorations rapportées concernant une taille de groupe *m* égale à quatre, nous estimons que les améliorations diminueront quand la taille de groupe augmentera. En effet, dans ce cas le nombre de tampons à traiter en calcul RS qui est de *m*-1, augmente par rapport au nombre de tampons à traiter en calcul XOR, qui est toujours 1.

7.4 Création d'un fichier

Dans ce qui suit, nous rapportons les temps de création d'un fichier $LH*_{RS}$, *k*-disponible ($k \in \{0, 1, 2\}$) dans différentes architectures de cases et les deux configurations matérielles. Les temps sont calculés sur le client.

Chaque expérimentation dans l'architecture SDDS2000 ou SDDS-TCP se déroule comme suit : un client procède à l'insertion de 25000 enregistrements (*b* = 10000, *b'* = 100), chacun de 100 octets. 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, et deux éclatements successifs de la case de données 0 qui crée la case de données 2, et de la case de données 1 qui crée la case de données 3. Notons que le client envoie une seule requête à la fois et attend l'accusé correspondant et qu'à la réception d'un ordre d'éclatement, une case de données est verrouillée jusqu'à fin éclatement. Ainsi, un éclatement implique l'arrêt des envois de requêtes d'insertion. Les accusés envoyés de la part des cases de données accusent réception et traitement de la requête d'insertion, mais aucun mécanisme d'acquittement n'est implanté entre les cases de données et les cases de parité. La dégradation des performances observées est due seulement à la propagation des mises à jour vers les cases de parité. La limite de notre travail est l'absence de garantie de réception des requêtes de mise à jour par les cases de parité.

Vu que les expérimentations se déroulent dans un réseau local et que la vitesse d'envoi de requêtes est freinée par la réception d'acquittement, nous notons un faible taux de perte de messages.

Rappelons que comparée à l'architecture SDDS2000, l'architecture SDDS-TCP améliore la gestion des connexions TCP. Ce qui affecte particulièrement le scénario d'éclatement, donc un transfert plus rapide d'enregistrements entre deux cases de données et également des tampons de mises à jours des cases de données vers les cases de parité.

En ce qui concerne l'architecture SDDS-Ack, son principal atout par rapport à l'architecture SDDS-Ack, est le fait qu'au niveau du client et des cases de données, a été ajoutée une couche de contrôle de flux et d'acquittement des requêtes d'insertion. Dans nos expérimentations nous avons fixé le crédit d'envoi du client à 5.

7.4.1 Architecture SDDS2000

La Figure 7-1 montre que le schéma LH^*_{RS} 1-disponible (k = 1) dégrade les performances de création d'un fichier de 35% par rapport au schéma LH^*_{RS} 0-disponible (LH^*_{LH}). Le coût de mis à jour d'une case de parité en plus que la première case de parité est de 13,34%. L'allure des courbes est plutôt linéaire, sauf au moment de l'éclatement d'une case de données, qui nécessite un temps en plus, en raison de l'attente des accusés.

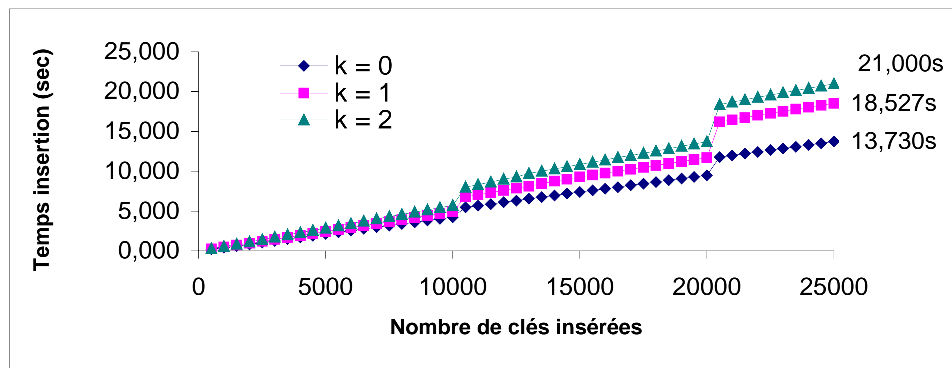


Figure 7-1: Performances de création d'un fichier d'un fichier LH^*_{RS} *k*-disponible -[*k* =0, 1, 2 ; Architecture SDDS2000, 1ére config.].

La Figure 7-2 montre le temps moyen d'insertion par enregistrement, tels que l'axe des abscisses montre les clés insérées et l'axe des ordonnées note le temps d'insertion de la clé en question. Remarquons que, avant le premier éclatement (clé < 10000) le temps moyen d'insertion est constant. Au moment du premier éclatement de la case de données 0, le client connaît un ralentissement et un arrêt des envois des requêtes d'insertion du coté du client, et ce jusqu'à arrivée de l'accusé de la clé qui subit l'éclatement. Après l'éclatement et jusqu'au deuxième et troisième éclatement respectivement de la case de données 0 et la case de données 1, qui se déroulent presque en même temps, les temps d'insertions des clés -(clé > 10000 et clé < 20000), après avoir enregistré une dégradation, elles retournent à la moyenne.

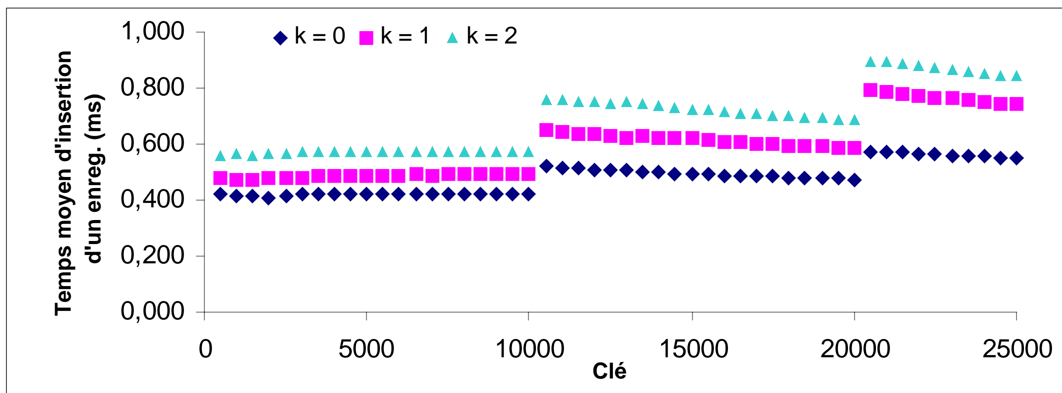


Figure 7-2: Temps moyen d'insertion d'un enregistrement dans un LH*$_{RS}$ *k*-disponible -[*k* = 0, 1, 2 ; Architecture SDDS2000, 1ére config.].

Notons que nous avons calculé les temps moyens d'insertion sur un intervalle de 500 insertions, et avec la formule suivante :

$$\frac{Temps\ Insertion\ Clé\ 'C+500' - Temps\ Insertion\ Clé\ 'C'}{500}$$

Les moyennes obtenues ne sont pas très significatives ; surtout sur les intervalles [10000, 10500] et [20000, 20500] qui subissent le coût de l'éclatement. En fait, seules quelques requêtes d'insertion subissent la pénalité de l'éclatement.
En conclusion, le temps requis pour l'insertion d'un enregistrement est constant, à l'exception du cas où la requête d'insertion coïncide avec l'éclatement d'une case de données. En effet, dans ce cas les requêtes sont mises en attente, car la case est verrouillée pour éclatement.

7.4.2 Architecture SDDS-TCP

7.4.2.1 Première Configuration

Sur une vingtaine d'expérimentations, les temps de création de fichier varient au point que nous n'avons pas pu conclure de façon formelle sur l'avantage d'utilisation du CG(2^{16}) et de la nouvelle matrice. Dans la suite, nous rapportons les moyennes des temps de création de fichier.
La Figure 7-3 montre que le schéma LH*$_{RS}$ 1-disponible (*k* = 1) dégrade les performances de création d'un fichier de 35% par rapport au schéma LH*$_{RS}$ 0-disponible (LH*$_{LH}$). Le coût de mise à jour d'une case de parité en plus que la première case de parité est de 12%. Nous avons obtenu des dégradations de la même échelle dans l'architecture SDDS2000 (§7.4.1).

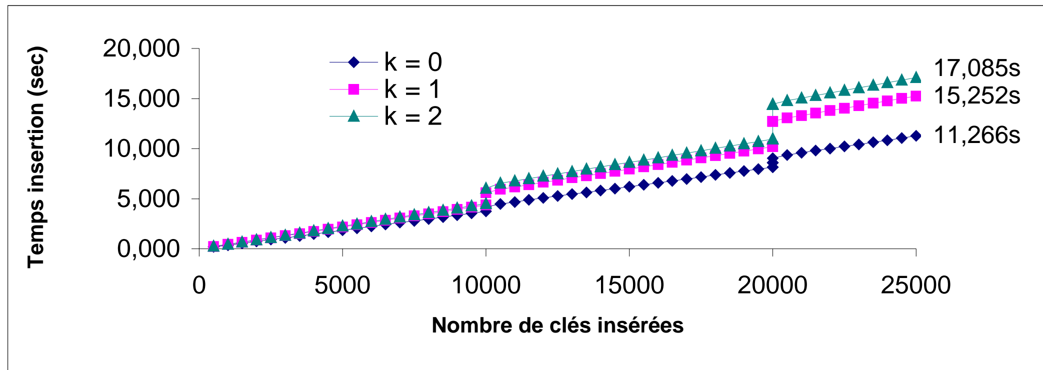


Figure 7-3: Performances de création d'un fichier LH*$_{RS}$ *k*-disponible -[*k* = 0, 1, 2 ; Architecture SDDS-TCP ; 1ére config.].

7.4.2.2 Deuxième Configuration

La Figure 7-4 montre que le schéma LH*$_{RS}$ 1-disponible (k = 1) dégrade les performances de création d'un fichier de 21% par rapport au schéma LH*$_{RS}$ 0-disponible (LH*$_{LH}$). Le coût de mise à jour d'une case de parité en plus que la première case de parité est en moyenne de 8%.

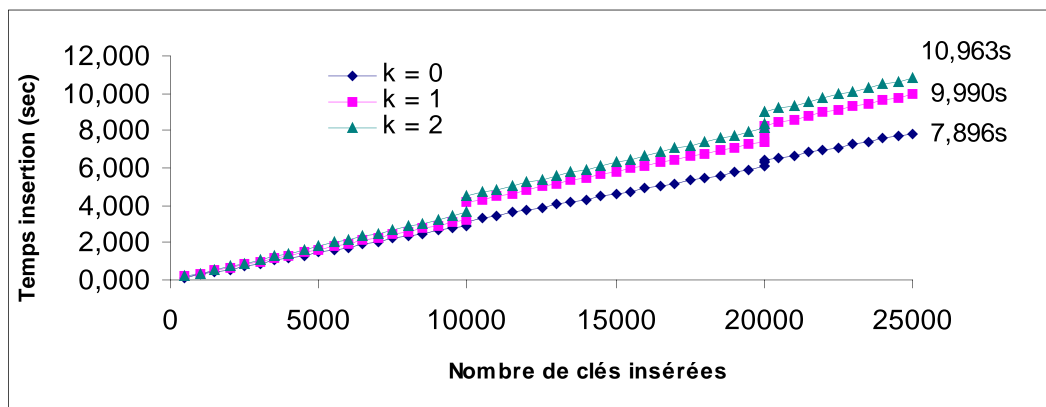


Figure 7-4: Performances de création d'un fichier LH*$_{RS}$ *k*-disponible -[*k* = 0, 1, 2 ; Architecture SDDS-TCP ; 2éme config.].

La Table 7-3 rapporte les moyennes des temps d'insertion des enregistrements, mesurées sur le client dès arrivée de la clé attendue. Le client enregistre le temps de réception de certaines clés particulières, dites *clés à risque* 10001, 20001 .. 20005. En effet, ces dernières risquent de subir le coût de l'éclatement. La connaissance du temps d'insertion de ces clés, permet de définir d'une part le délai max. (ang. *time-out*), et d'autre part de mesurer l'impact de l'éclatement sur les performances d'insertions.

Ack Clef	*k = 0*	*k = 1*	*K = 2*
500	0,282	0,344	0,342
9000	0,294	0,314	0,376
9500	0,282	0,342	0,344
10000	0,312	0,314	0,374
10001	219,000	890,000	907,000
10500	0,313	0,315	0,375

11000	0,282	0,312	0,344
11500	0,280	0,344	0,374
20000	0,290	0,312	0,376
20002	219,000	0,370	0,405
20003	0,303	250,000	297,000
20004	0,314	625,000	594,000
20005	93,000	0,414	0,450
20500	0,317	0,347	0,378
21000	0,312	0,344	0,376
21500	0,282	0,312	0,374
22000	0,312	0,344	0,376
24500	0,294	0,344	0,344
25000	0,312	0,376	0,374

Table 7-3: Temps d'insertion d'un enregistrement (ms) dans un LH*$_{RS}$ *k*-disponible -[*k* = 0, 1, 2 ; Architecture SDDS-TCP ; 2éme config.].

7.4.2.3 Comparaison

En comparant les performances de la création de fichier avec une architecture de cases SDDS-TCP dans la première configuration à celles la création de fichier avec une architecture de cases SDDS-TCP dans la deuxième configuration, nous en concluons que la deuxième configuration (*i*) améliore les performances de création d'un fichier LH*$_{RS}$ de 38,64% et (*ii*) diminue le coût de transfert de mises à jour vers les cases de parité (le coût de mise à jour de la première case de parité est de 35% dans la première config. vs. 21% dans la deuxième config. et le coût de mise à jour de la deuxième case de parité 12% dans la première config. vs. 8% dans la deuxième config.).

7.4.3 Architecture SDDS-Ack

Faute de temps, seule la stratégie d'acquittement et de contrôle de flux entre [Client – Case de Données], décrite en §5.4.4, a été implantée et évaluée. La stratégie d'acquittement des messages de mises à jour par les cases de parité, décrite en §5.4.5, a été partiellement implantée.

La Figure 7-5 montre les performances de création d'un fichier LH*$_{RS}$ *k*-disponible, tel que k = 0,1, 2. Coté client, le paramètre *fenêtre* est égal à 5. 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*$_{RS}$ 0-disponible (LH*$_{LH}$). Le coût de mise à jour d'une case de parité en plus que la première case de parité est en moyenne de 11%.

Nous avons tenu à estimer le taux de perte des de mises à jour entre [Case de données – Case de parité]. Ainsi, nous avons implanté des compteurs de messages de mises à jour envoyés par les cases de données et d'autres au niveau des cases de parité pour compter le nombre de messages de mises à jour reçus. 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.

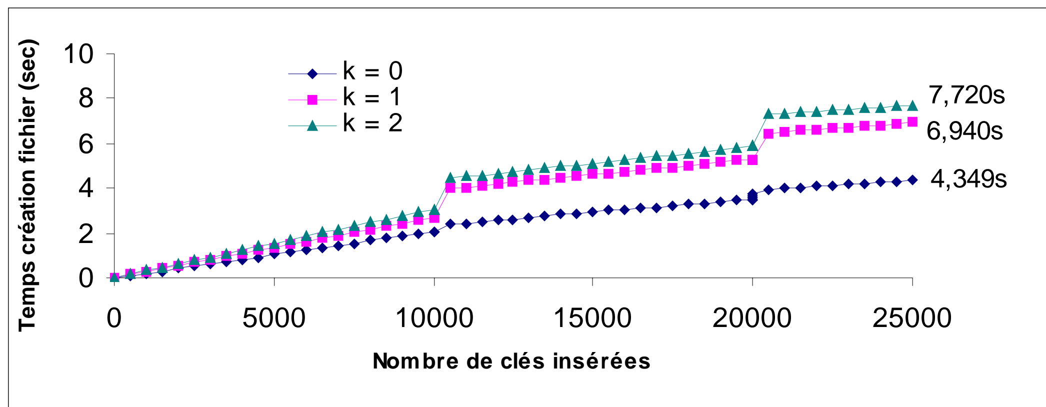


Figure 7-5: Performances de création d'un fichier LH*RS *k*-disponible -[*k* = 0, 1, 2 ; Architecture SDDS-Ack ; 2éme config.].

7.4.4 Conclusion

En comparant les performances de la création de fichier avec une architecture de cases SDDS2000 (§7.4.1) à celles la création de fichier avec une architecture de cases SDDS-TCP (§7.4.2), nous en concluons que l'architecture SDDS-TCP améliore les temps de création de fichier en moyenne de 18%, et ce grâce à une meilleure gestion des connexions TCP. L'amélioration sur la composante réseau (temps de communication) étant concernée par l'apport de SDDS-TCP, doit être supérieure à 18%.

Les résultats obtenus sont raisonnables, la dégradation des performances de passage d'un schéma *k*-disponible vers un schéma *k*+1-disponible ($k \neq 0$), doit être inférieure à la dégradation due à un passage d'un schéma 0-disponible vers un schéma 1-disponible. Dans le dernier cas, la dégradation est due à la préparation des tampons de mise à jour et leurs envois à la première case de parité. Par contre, par rapport au dernier cas, le passage d'un schéma 1-disponible vers un schéma 2-disponible, ne nécessite que l'envoi supplémentaire de mises à jour à la deuxième case de parité.

7.5 Requêtes de Recherche

Nous évaluons les requêtes de recherche en parallèle et les requêtes de recherche en synchrone, et en mode non dégradé.

Pour les requêtes de recherche en parallèle, le client envoie des requêtes de recherche par clef en parallèle à quatre cases de données. Telle que la clef est dans l'intervalle [1, *x*], *x* variant de 10 000 à 100 000. Ainsi, chaque serveur reçoit $x/4$ requêtes de recherche, et renvoie une réponse à chaque requête dans un message individuel.

Pour les requêtes de recherche en synchrone, le client envoie un total de x requêtes de recherche par clef, mais une seule requête à la fois, et se met en attente de la réponse correspondante, avant d'envoyer la requête suivante.

Dans ce qui suit, nous rapportons les mesures de performances des requêtes de recherche dans un fichier de 125 000 enregistrements (b = 50000, b' = 100), réparti sur quatre cases de données, et ce dans chacune des configurations matérielles.

7.5.1 Première Configuration

La Figure 7-6 montre que les temps de recherche par enregistrement sont indépendants du nombre de requêtes. En effet, notons un temps moyen de recherche de 0,34ms par requête synchrone, et un temps moyen de 0,084ms pour requête parallèle. Pour les requêtes parallèles, les quatre serveurs de données ont un débit de 11834 requêtes/sec mesuré sur le client.

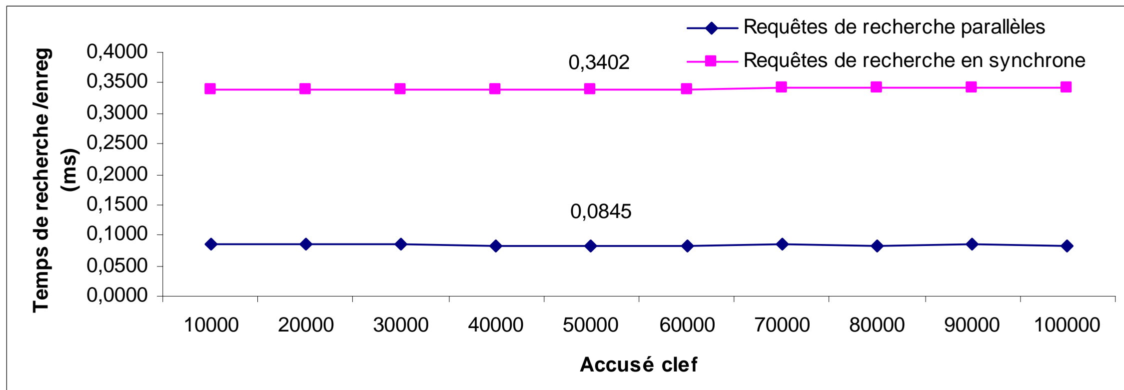


Figure 7-6: Performances des requêtes de recherche dans la 1ére configuration.

7.5.2 Deuxième Configuration

La Figure 7-7 montre que les temps de recherche par enregistrement sont indépendants du nombre de requêtes. En effet, notons un temps moyen de recherche de 0,24ms par requête synchrone, et un temps moyen de 0,056ms pour requête parallèle. Pour les requêtes parallèles, les quatre serveurs de données ont un débit de 17857 requêtes/sec mesuré sur le client.

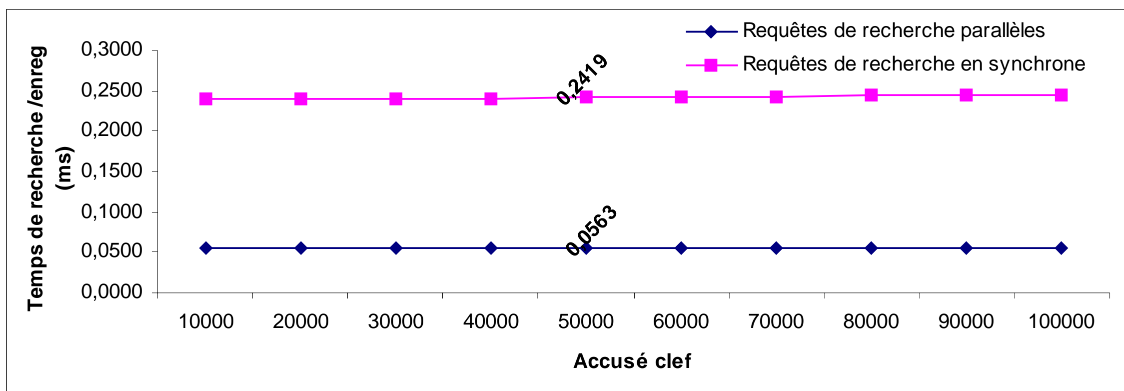


Figure 7-7: Performances des requêtes de recherche dans la 2éme configuration.

#Req. recherche (x)	*Débit Envoi Client (req./sec)*	*Débit Réception Client (req./sec)*	*% de perte*
5000	40000	10309	0,0036
10000	42553	10660	0,0045
20000	41322	10157	0,0078
30000	41724	9696	0,0127

Table 7-4: Analyse des performances de requêtes parallèles ciblant un seul serveur de données dans la 2éme Configuration.

Pour les requêtes parallèles, nous nous sommes intéressés au débit d'un seul serveur de données, ainsi nous avons mené des expérimentations consistant en l'envoi par le client de x requêtes à un seul serveur de données, $x \in \{5000, 10000, 20000, 30000\}$. Les mesures de performances sont récapitulées dans la Table 7-4.

Remarquons que le taux de perte de messages (requêtes ou réponses) augmente avec le nombre de requêtes envoyées, et que le débit d'envoi de requêtes et de réception de réponses côté client est maximal pour x égal à 10 000.

En moyenne, le temps de réponse à une requête de recherche est égal à 0,098ms. Il comprend le temps d'envoi de la requête plus le temps d'envoi de la réponse –donc approximativement le RTT dans le réseau (ang. *Round Trip Time*), plus le temps de traitement d'une requête par le serveur.

Rappelons que le client est doté d'une file dans laquelle les réponses aux requêtes sont empilées et d'un seul *thread* de traitement des réponses. Alors qu'un serveur de données est doté de quatre *threads* de travail pour traiter les requêtes reçues. Les mesures de performance prouvent que les quatre *threads* de travail du serveur de données infèrent un parallélisme intra-requêtes. D'ailleurs, il serait intéressant de déterminer le paramètre T -le nombre de *threads* dans un serveur au-delà duquel les performances d'un serveur se dégradent [WGBC00] décrit en §2.4.2.

7.5.3 Conclusion

Par rapport à la première configuration, l'amélioration des performances dans la deuxième configuration englobe l'amélioration du temps de communication et du temps de traitement. La deuxième configuration améliore les temps de recherche de 29% pour les recherches parallèles et de 33,4% pour les recherches en synchrone.

7.6 Récupération d'enregistrements de données

La récupération d'enregistrements de données est exécutée en cas de recherche en mode dégradé. En cas de détection d'un échec, nous avons le choix soit de procéder à la récupération de toutes les cases en échec du groupe, soit à la récupération de seulement des cases en échec ou la récupération d'un ensemble précis d'enregistrements. Tout dépend des performances de récupération, et de la demande du client.

7.6.1 Description des expérimentations

La procédure "*Récupérer un enregistrement de données*", implantée au niveau de la case de parité, désignée pour récupérer un enregistrement, s'exécute comme-suit,

1. Rechercher la clef de l'enregistrement dans la structure de la case de parité,
2. Envoyer des requêtes de recherche vers les cases disponibles, il s'agit au maximum de m-1-v requêtes de recherche, tel que v désigne le nombre d'enregistrements factices du groupe de parité.
3. Attendre jusqu'à réception des réponses aux requêtes envoyées,
4. Récupérer l'enregistrement manquant,
5. Envoyer l'enregistrement récupéré au client.

Rappelons que la case de parité, chargée de récupérer un enregistrement de données de clef k, appartenant à la case de données d, effectue une recherche séquentielle coûteuse de l'enregistrement de données en $O(b)$ si la case contient b enregistrements. Le but de la recherche, est de trouver l'enregistrement de parité, dont le champ *ListeClefs* à la position (d mod m) contient la clef k.

Nous avons mené une série d'expérimentations, consistant en première étape en la création d'un fichier LH^*_{RS} de 125 000 enregistrements (b = 50000) avec 31250 enregistrements par case et 1-disponible. En deuxième étape, nous simulons l'échec d'une case de données et récupérons x enregistrements de données appartenant à la case de données en échec. Nous varions l'algorithme de décodage, XOR ou RS dans le corps de Galois $CG(2^8)$ ou $CG(2^{16})$. Le choix entre un décodage XOR et RS est prépondérant de la disponibilité de la première case de parité.

Les résultats rapportés dans les sous-sections suivantes indiquent le temps nécessaire à la récupération d'un enregistrement de données, seulement au niveau de la case de parité. En effet, les délais d'attente de réponse au niveau du client et le pré-traitement effectué par le coordinateur particulièrement le processus de détection d'échec, ne sont pas pris en compte. Ces temps requièrent un paramétrage judicieux en fonction du RTT dans le réseau, temps réponse serveurs etc.…

Dans ce qui suit, nous rapportons les temps de récupération d'un enregistrement par une case de parité. Nous avons dû mesurer la récupération de plusieurs enregistrements pour estimer le temps de récupération d'un seul enregistrement.

7.6.2 Première Configuration

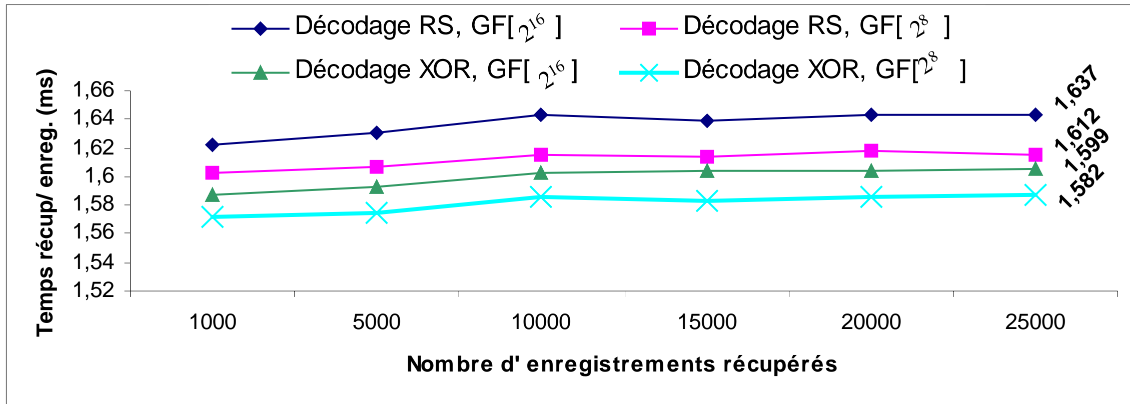


Figure 7-8: Récupération de *x* enregistrements de données dans la première configuration, par décodage [RS, CG(2^{16})], [RS, CG(2^8)], [XOR, CG(2^{16})],[RS, CG(2^8)].

La Figure 7-8 montre que les résultats sont linéaires et indépendants de *x* –le nombre d'enregistrements récupérés. le temps de récupération pour les petites valeurs de *x* est inférieure à la moyenne, et plus *x* se rapproche de 31250 plus ils se stabilisent, ceci est du aux conditions de l'expérimentation. Dans l'expérimentation, nous générons *x* clefs d'enregistrements appartenant à la case de données en échec, ceci fait que si l'enregistrement se trouve dans les premiers rangs dans la case de parité il sera vite récupéré. En effet, n'oubliant pas que la procédure de décodage effectue un parcours séquentiel à la recherche de la clef de l'enregistrement de données.

Remarquons que :

* Le corps de Galois CG(2^{16}), par rapport au corps CG(2^8), dégrade les performances de 1,55% -en cas d'un décodage RS, et de 1,07 % -en cas d'un décodage XOR.

* L'utilisation d'un décodage XOR au lieu d'un décodage RS, améliore le temps de récupération de 1,86% dans le corps CG(2^8), et de 2,32% dans le corps CG(2^{16}). Cette amélioration globale prouve un gain en décodage XOR.

7.6.3 Deuxième Configuration

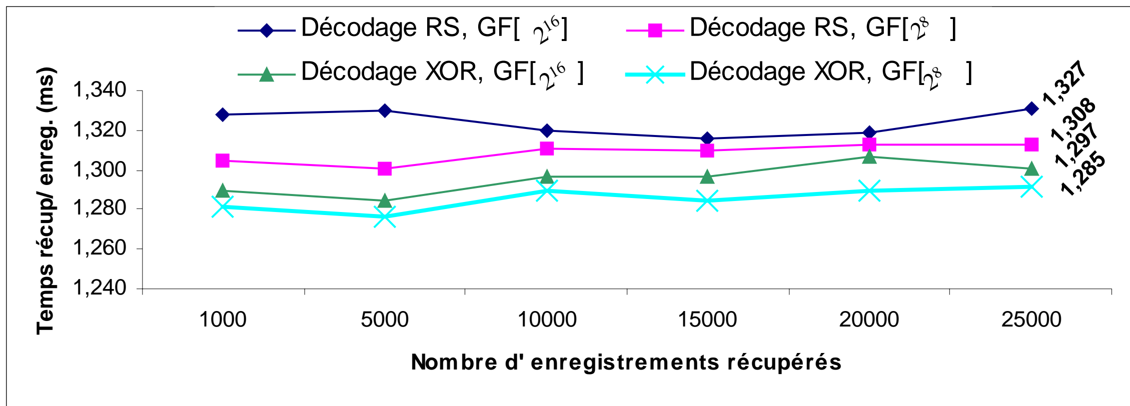


Figure 7-9: Récupération d'enregistrements de données dans la deuxième configuration, par décodage [RS, CG(2^{16})], [RS, CG(2^8)], [XOR, CG(2^{16})],[RS, CG(2^8)].

Remarquons que :

* Le corps de Galois CG(2^{16}), par rapport au corps CG(2^8), dégrade les performances de 1,43% -en cas d'un décodage RS, et de 0,93 % -en cas d'un décodage XOR.
* L'utilisation d'un décodage XOR au lieu d'un décodage RS, améliore le temps de récupération de 1,76% dans le corps CG(2^8), et de 2,26% dans le corps CG(2^{16}).

Des améliorations du même ordre ont été obtenues dans la première configuration (voir §7.6.2).

Nous avons mesuré le temps de détermination le rang d'un enregistrement de données dans une case de parité sachant sa clef et l'indice dans le groupe de la case de données à laquelle il appartient, nous l'estimons à 0,822ms. Il représente plus que 60% du temps de récupération. Pour diminuer le temps de recherche, il faudrait implanter un indexe au niveau des cases de parité par case de données, qui connaissant la clef d'un enregistrement de données, nous déterminons son rang (voir §8.2.2).

7.6.4 Comparaison

Notons que, aussi bien dans la première configuration, que dans la deuxième configuration, (*i*) le temps de récupération d'un enregistrement dans CG(2^8) est inférieur donc meilleur que dans CG(2^{16}) et que (*ii*) le décodage XOR par rapport au décodage RS améliore les performances de récupération. Par rapport au premier évoqué, du point de vue implantation il existe une seule différence, et ce, vu que les cases disponibles convertissent par casting une chaîne de symboles de CG(2^{16}) en une chaîne de caractères pour l'inclure dans un message. La case de parité aussi reconvertit la chaîne de caractères en chaîne de symboles appartenant à CG(2^{16}). Enfin, une conversion en chaîne de caractères est faite pour envoyer l'enregistrement au client.

La deuxième configuration améliore le temps de récupération d'un enregistrement, par rapport à la première configuration de 19%. En comparant cette amélioration à celle de l'ordre de 30% obtenue pour la recherche d'un enregistrement (voir §7.5.3), c'est sans doute du à la disparité des améliorations relatives aux composantes réseau et vitesse de traitement. En effet, il y a lieu de noter que le temps CPU est dominant dans le scénario de récupération d'un enregistrement de données, et est plus grand que le temps CPU de recherche d'un enregistrement de données. Ceci implique une moindre amélioration globale.

7.7 Requêtes de Mise à Jour Aveugles

7.7.1 Description des Expérimentations

Le scénario de mise à jour se déroule comme suit, à partir du client des requêtes de mises à jour aveugles d'enregistrements de la même case de données sont envoyées. Dans le but d'estimer le coût des mises à jour, et la dégradation des performances due à la répercussion des mises à jour aux cases de parité, chaque expérimentation consiste à créer un fichier k-disponible, $k \in \{0, 1, 2, 3\}$.

Le fichier crée est de 10 000 enregistrements et s'étend sur une seule case de données. A la réception d'une requête de mise à jour, la case de données répercute cette mise à jour sur l'ensemble de k cases de parité. Elle émet alors k requêtes de mises à jour et doit attendre impérativement les k accusés des requêtes de mises à jour émises, provenant des k cases de parité, pour enfin acquitter la requête de mise jour auprès du client.

Nous avons conçu deux schémas de mises à jour. Le premier schéma *Mises à Jour Sans Acks*, consiste en l'envoi de requêtes de mises à jour vers une case de données. Cette dernière répercute les mises à jour sur les k cases de parité de son groupe. Le scénario se déroule sans attente d'accusés et suppose que le réseau est fiable. Alors que le deuxième schéma *Mises à Jour avec Attente Acks* traite les accusés, en effet les requêtes de mises à jour sont d'abord acquittées auprès de la case de données pour les k cases de parité impliquées, enfin auprès du client pour la case de données.

Le nombre de requêtes de mises à jour est dans l'ensemble {500, 1000, 5000, 8000} et les expérimentations concernent les deux corps de Galois : CG(2^8) et CG(2^{16}).

7.7.2 Performances de MAJs

Nous n'avons évalué les requêtes de mise à jour aveugles que dans la deuxième configuration, car l'expérimentation a été faite après que le laboratoire s'est dotée de la nouvelle configuration matérielle.

# Requêtes de MAJ	Sans Acks (ms)	k = 0 (ms)	k = 1 (ms)	k = 2 (ms)	k = 3 (ms)
500	0,030	0,250	0,470	0,562	0,574
1000	0,047	0,250	0,488	0,559	0,581
5000	0,041	0,255	0,489	0,569	0,588
8000	0,041	0,259	0,497	0,573	0,597

Table 7-5: Temps de mise à jour d'un enregistrement de données et de *k* enregistrements de parité, dans un fichier LH*$_{RS}$, *k*-disponible, dans CG(2^8).

# Requêtes de MAJ	Sans Acks (ms)	k = 0 (ms)	k = 1 (ms)	k = 2 (ms)	k = 3 (ms)
500	0,030	0,250	0,488	0,544	0,576
1000	0,047	0,250	0,484	0,547	0,572
5000	0,041	0,256	0,501	0,557	0,584
8000	0,041	0,258	0,508	0,563	0,592

Table 7-6: Temps de mise à jour d'un enregistrement de données et de *k* enregistrements de parité, dans un fichier LH*$_{RS}$, *k*-disponible, dans CG(2^{16}).

Remarquons que,

* Le deuxième schéma *Mises à Jour avec Attente Acks* implique un temps de mise à jour d'un enregistrement de données au moins six fois supérieur au temps de mise à jour du schéma *Mises à Jour Sans Acks*.
* L'attente de l'accusé de la part de la première case de parité pour un fichier 1-disponible, double le temps de mise à jour. Par contre, dans un schéma *k*-disponible, tel que $k > 1$, le temps de mise à jour par rapport au schéma (*k-1*)-disponible, augmente le temps de mise à jour de 5% à 20%. En effet, notons que de k = 1 à 2, une dégradation en moyenne de 16% pour CG(2^8) et en moyenne de 11% pour CG(2^{16}) ; et que de k = 2 à 3, une dégradation en moyenne de 4% pour CG(2^8) et en moyenne de 5% pour CG(2^{16}). La dégradation est plus importante de k = 1 à 2 que k = 2 à 3, car pour k égal à 1 la mise à jour est plus rapide puisqu'une mise à jour XOR est exécutée. Notons aussi que la dégradation de k = 1 à 2 est moins importante dans CG(2^{16}) que dans CG(2^8), ceci s'explique par le fait que le XOR est moins effectif dans CG(2^{16}) que dans CG(2^8).
* Le temps de mise à jour d'un enregistrement varie peu, entre les corps de Galois CG(2^8) et CG(2^{16}).
* Les performances de mises à jour sont meilleures que les performances d'insertion rapportées dans la Table 7-3. Ces dernières subissent les éclatements internes de cases LH.

7.8 Création d'une Case de Parité

Le scénario d'augmentation de la haute disponibilité, décrit en §6.4, ajoute une case de parité à un groupe. Nous décrivons brièvement le déroulement et les conditions des expérimentations, puis nous rapportons le temps de création d'une case de parité, pour différentes architectures des cases, différentes configurations, différents algorithmes de codage, et différentes tailles de cases.

Nous avons évalué les performances du scénario de création d'une case de parité se basant sur UDP -Scénario 6-10, dans [M00], les expérimentations ont démontré des gains en performance et fiabilité en faveur du scénario se basant sur TCP -Scénario 6-11.

7.8.1 Description des Expérimentations

Dans chaque expérimentation, nous créons un fichier LH*$_{RS}$ de 2,5*b enregistrements de données. Ce dernier s'étend sur quatre cases de données, tel que chaque case contienne exactement 0,625*b enregistrements de données.

Le *Temps Total* de création d'une case de parité comprend le *Temps de Communication* et le *Temps de Traitement*. Le *Temps de Communication* est le temps nécessaire pour établir des connexions TCP/IP afin de recevoir le contenu des cases de données. Tandis que le *Temps de Traitement* désigne le temps de traitement des tampons reçus des cases de données et inclut le temps de codage. La différence entre *Temps Total* et *Temps de Communication* + *Temps de Traitement* est du aux commutations des *threads* (ang. *thread switch context*).

Dans ce qui suit, nous rapportons les résultats relatifs à la création d'une case de parité dans deux architectures différentes, respectivement l'architecture SDDS2000 (§5.2), et l'architecture SDDS-TCP (§5.3).

7.8.2 Architecture des Cases SDDS2000

La Table 7-7 rapporte les résultats des expérimentations réalisées dans la première configuration, avec des cases de données et de parité dont l'architecture interne est SDDS2000, et un codage RS dans le corps de Galois CG(2^8).

* b	* Temps Com.(s)	* Temps Trait. (s)	* Total (s) *
5000	2,211	0,302	2,523
10000	2,433	0,611	3,044
25000	3,185	1,652	4,847
50000	4,667	3,395	8,062

Table 7-7: Temps de création d'une case de parité, Architecture SDDS2000, codage RS dans CG(2^8).

Notons que le temps nécessaire pour établir des connexions TCP/IP avec les quatre cases de données est supérieur au temps de traitement. En effet, le temps de communication représente {87.63%, 79.93%, 65.71%, 57.89%} du temps total, et ce respectivement aux tailles de cases de {5000, 10000, 25000, 50000}. Ainsi, il s'avère plus intéressant dans ce cas de choisir une grande taille de case, pour amortir le coût des connexions TCP/IP, et ce partant de la constatation : plus la taille de cases est grande, moins le temps de communication influe sur le temps total.

Nous notons une linéarité des temps de traitement. En effet, le temps de traitement dépend de *b* la taille des cases. Les temps de codage obtenus sont de {1.035, 1.023, 0.946, 0.920} MO/sec respectivement pour les tailles de cases {0.3125, 0.625, 1.5625, 3.125}MO. La légère dégradation des performances de codage est due à la manipulation de tampons plus grands.

En ce qui concerne les temps de communication, notons que de $b = 5000$ à $b = 10000$, la taille des tampons envoyés à la case de parité double, mais une augmentation de seulement 10% est observée. Quand la taille des tampons quintuple, une augmentation de 44% est observée. Ce qui montre le temps de transfert d'un tampon par TCP/IP dépend à la fois de la taille du tampon et du temps d'accès au site distant dit encore temps de latence.

7.8.3 Architecture des Cases SDDS-TCP

Avec l'architecture SDDS-TCP, nous avons mesuré les temps de création d'une case de parité par le Scénario 6-11, dans les deux configurations matérielles. Pour chaque configuration matérielle, nous avons varié l'algorithme de codage, étant {[XOR,

$CG(2^8)$], [XOR, $CG(2^{16})$], [RS^{++}, $CG(2^8)$], [RS^{++}, $CG(2^{16})$]}et *b* étant la taille des cases, $b \in \{5000, 10000, 25000, 50000\}$.

Rappelons que RS^{++} désigne la création d'une case de parité selon une colonne de la matrice optimisée, comportant un '1' et *m*-1 coefficients éléments d'un corps de Galois avec un pré-calcul du *log*.

7.8.3.1 Première configuration

La Table 7-8 montre les résultats obtenus pour la création d'une case de parité.

Les cases de parité se génèrent selon un codage RS^{++}, hormis la première case de parité de chaque groupe de parité. En effet, grâce à la $m^{éme}$ colonne de '1's de la matrice génératrice, la première case de parité peut être créée ou récupérée en utilisant un codage XOR. En comparaison avec un codage RS^{++}, un codage XOR réalise des gains en performance en temps de traitement de 20% dans le corps de Galois $CG(2^8)$ et de 8% dans $CG(2^{16})$. Notons également que :

* Un codage XOR dans le corps $CG(2^{16})$ dégrade les performances en temps de traitement de 9% par rapport à un codage XOR dans $CG(2^8)$.
* Un codage RS^{++} dans le corps $CG(2^{16})$ améliore les performances en temps de traitement de 4,5% par rapport à un codage RS^{++} dans $CG(2^8)$.

Corps de Galois: $CG(2^8)$

b	Codage XOR Tot. (s)	Codage XOR Trait. (s)	Codage XOR Com. (s)	Codage RS^{++} Tot. (s)	Codage RS^{++} Trait. (s)	Codage RS^{++} Com. (s)
5000	1,326	0,172	1,112	0,589	0,212	0,337
10000	1,991	0,335	1,590	1,198	0,434	0,705
25000	2,598	0,937	1,526	2,155	1,135	0,875
50000	3,803	1,905	1,624	4,200	2,363	1,568

Corps de Galois : $CG(2^{16})$

b	Codage XOR Tot. (s)	Codage XOR Trait. (s)	Codage XOR Com. (s)	Codage RS^{++} Tot. (s)	Codage RS^{++} Trait. (s)	Codage RS^{++} Com. (s)
5000	0,679	0,186	0,460	0,377	0,202	0,143
10000	1,124	0,377	0,687	0,773	0,424	0,291
25000	2,121	1,031	0,950	2,177	1,134	0,907
50000	3,745	2,055	1,420	3,778	2,107	1,400

Table 7-8: Performances de création d'une case de parité, pour différentes tailles de cases, et différents algorithmes de codage, dans la 1ére config.

Lors des expérimentations, nous avons noté une variation des temps de communication. A titre d'exemple, pour *b* = 5000, le temps minimum de communication obtenu pour la création d'une case de parité est 0,14sec et le temps maximum est 2,324sec. Cette variation concerne en particulier les tailles de cases inférieures à 50000, s'accentue plus la taille des cases *b* est petite.

7.8.3.2 Deuxième Configuration

Ci-dessous, la Table 7-9 montre les résultats obtenus pour la création d'une case de parité.

Corps de Galois: CG(2^8)

* b *	Codage XOR Tot. (s)	Trait. (s)	Com. (s) *	Codage RS++ Tot. (s)	Trait. (s)	Com. (s) *
5000	0,153	0,109	0,031	0,184	0,140	0,019
10000	0,356	0,256	0,056	0,426	0,336	0,059
25000	0,922	0,638	0,150	1,085	0,816	0,156
50000	1,703	1,265	0,314	2,058	1,618	0,324

Corps de Galois : CG(2^{16})

* b *	Codage XOR Tot. (s)	Trait. (s)	Com. (s) *	Codage RS++ Tot. (s)	Trait. (s)	Com. (s) *
5000	0,190	0,140	0,029	0,193	0,149	0,035
10000	0,429	0,304	0,066	0,446	0,328	0,059
25000	0,838	0,626	0,150	0,870	0,650	0,157
50000	1,711	1,281	0,305	1,781	1,336	0,316

Table 7-9: Performances de création d'une case de parité, pour différentes tailles de cases, et différents algorithmes de codage, dans la 2éme config.

Les cases de parité se génèrent selon un codage RS^{++}, hormis la première case de parité de chaque groupe de parité. En effet, grâce à la *m*^{éme} colonne de '1's de la matrice génératrice, la première case de parité peut être créée ou récupérée en utilisant un codage XOR. En comparaison avec un codage RS^{++}, un codage XOR réalise des gains en performance en temps de traitement en moyenne de 22% dans le corps de Galois CG(2^8), et de 5% dans CG(2^{16}).

Notons également que :

* Un codage XOR dans le corps CG(2^{16}) dégrade les performances en temps de traitement de en moyenne de 15% par rapport à un codage XOR dans CG(2^8).
* Un codage RS^{++} dans le corps CG(2^{16}) améliore les performances en temps de traitement de 5% par rapport à un codage RS^{++} dans CG(2^8).

Contrairement à la première configuration, les temps de communication obtenus varient peu, avec le réseau Ethernet 1Gbps.

7.8.3.3 Conclusion

Notons que le temps de traitement représente en moyenne 74% du temps total. En conclusion, les améliorations réalisées en temps de communication grâce au réseau 1Gbps, et en temps de traitement correspondent à nos attentes. Par rapport à la première configuration, la deuxième configuration améliore en moyenne le temps de traitement de 28%, le temps de communication de 80% et globalement le temps total de création d'une case de parité de 55%.

7.8.4 Comparaison des 2 Architectures

Afin de comparer les résultats obtenus pour chacune des architectures, nous confrontons les résultats obtenus dans l'architecture SDDS2000 à ceux obtenus dans l'architecture *SDDS-TCP*. Pour cette dernière, nous considérons les expérimentations menées dans la première configuration et correspondant au codage [RS^{++}, CG(2^8)], nous nous intéressons qu'à l'amélioration du temps de communication, car l'architecture *SDDS-TCP* n'a pas de conséquence sur le temps de traitement.

L'architecture *SDDS-TCP* améliore les temps de communication de création d'une case de parité de 74%.
Remarquons que, les résultats s'inversent. En effet, contrairement à la première configuration, dans la deuxième configuration la composante réseau ne domine plus le temps total.

7.9 Récupération de Cases de Données

Dans ce qui suit, nous commençons par décrire les conditions et les paramètres d'expérimentation, puis nous rapportons et analysons les performances de récupération de f cases de données, $f \in \{1, 2, 3\}$, et ce versus les architectures de cases SDDS2000 et SDDS-TCP, les protocoles UDP et TCP et différents algorithmes de décodage XOR, RS et RS^+ dans les deux corps de Galois respectivement $CG(2^8)$ et $CG(2^{16})$.

7.9.1 Description des Expérimentations

Afin d'évaluer notre scénario de récupération, nous créons un fichier LH^*_{RS} de 125000 enregistrements ($b = 50000$) et k-disponible avec $k \in \{1, 2, 3\}$. Le fichier se crée sur quatre cases de données, tel que chaque case contient 31250 enregistrements de 100 octets, donc a un volume de 3,125MO. A chaque expérimentation, nous simulons l'échec de f cases de données tel que $f = k$, puis nous les récupérons.

Pour les scénarios utilisant le protocole TCP, Nous varions la taille du paramètre *tranche*, représentant le nombre d'enregistrements récupérés par itération de chaque case de données en échec. Et ce, afin de voir la corrélation entre le temps de récupération global et la valeur du paramètre *tranche*. L'ensemble des valeurs que peut prendre *tranche* est {1250, 3125, 6250, 15625, 31250}, représentant respectivement {4%, 10%, 20%, 50%, 100%}de la taille d'une case en nombre d'enregistrements. Le choix de la valeur du paramètre *tranche* doit être judicieux, en plus de notre analyse du scénario (§6.5.5), nous montrerons par expérimentation que des petites valeurs de *tranche* dégradent les performances.

Matrice	*Temps Inversion (µs)*
4×4	30
8×8	187,66
16×16	1401,21

Table 7-10: Temps d'inversion d'une matrice $m \times m$ en µs.

7.9.2 Architecture de Cases SDDS2000

L'ensemble des expérimentations rapportées dans cette section, ont été réalisées dans la première configuration, selon un décodage RS sans l'optimisation de pré-calcul du *log* et dans le corps de Galois $CG(2^8)$. Deux scénarios sont évalués, le scénario de récupération utilisant UDP -Scénario 6-15, et le scénario de récupération utilisant le protocole TCP/IP -Scénario 6-16.

7.9.2.1 Scénario de Récupération par UDP

Le scénario est décrit en §6.5.4. Nous avons mené une série d'expérimentations pour différentes tailles de cases $b \in \{5000, 10000, 25000, 50000\}$. Un fichier de 2,5*$b$ enregistrements est crée, pour avoir un nombre d'enregistrements par case est égal à b*0,625.

#Enregs/ case	Temps récup. (s)	Temps/ iter (ms)	Temps/ enreg (ms)
Récupération d'1 case de données			
3125	1,742	0,557	0,557
6250	3,465	0,554	0,554
15625	8,662	0,554	0,554
31250	17,375	0,556	0,556
Récupération de 2 cases de données			
#Enregs/ case	*Temps récup. (s)*	*Temps/ iter (ms)*	*Temps/ enreg (ms)*
3125	2,043	0,654	0,327
6250	4,076	0,652	0,326
15625	10,185	0,652	0,326
31250	20,349	0,651	0,325
Récupération de 3 cases de données			
#Enregs/ case	*Temps récup. (s)*	*Temps/ iter (ms)*	*Temps/ enreg (ms)*
3125	2,564	0,820	0,273
6250	4,797	0,767	0,256
15625	11,928	0,763	0,254
31250	23,895	0,765	0,255

Table 7-11: Performances de récupération de f cases de données, f = 1, 2, 3 - [Taille récupérée f*3,125MO ; UDP ; décodage RS ; CG(2^8); SDDS2000 ; 1ére config.].

Les résultats obtenus prouvent que le temps de récupération d'un enregistrement est indépendant de la taille d'une case. Un temps constant de 0,10 ms est requis pour récupérer un enregistrement supplémentaire de 100 octets par itération.

Notons que le temps de récupération d'un enregistrement décroît plus le nombre f de cases de données en échec à récupérer augmente. Ceci est du à la factorisation des messages d'interrogation des cases disponibles. En effet, à chaque itération, nous interrogeons au maximum m-1 cases pour récupérer f enregistrements. Notons des vitesses de récupération de f cases de données indépendantes de b, de 0,18 MO/sec pour f = 1, de 0,30 MO/sec pour f = 2 et de 0,40 MO/sec pour f = 3.

Malgré la scalabilité prouvée du scénario et la linéarité des résultats obtenus pour différentes tailles de cases, nous estimons que le scénario de récupération basé sur le protocole UDP devient coûteux plus b augmente. Le scénario basé sur le protocole TCP/IP, pallie les majeurs inconvénients du scénario basé sur le protocole UDP, et apporte de meilleures performances pour de grandes tailles de cases.

7.9.2.2 Scénario de Récupération par TCP

Le scénario est décrit en §6.5.5. Nous avons mené une série d'expérimentations de récupération de f cases de 31250 enregistrements f = 1,2,3 ; par *tranche* de {1250, 3125, 6250, 15625, 31250} enregistrements. Le fichier LH*$_{RS}$ (N = 125 000) s'étend sur quatre cases de données.

Récupération d'une case de données			
Tranche	*Temps Tot. (s)*	*Temps Com. (s)*	*Temps Trait. (s)*
1250	50,352	46,968	1,161
3125	23,073	20,008	1,052
6250	13,650	10,523	1,043
15625	8,292	4,986	1,042
31250	6,700	3,134	1,042
Récupération de 2 cases de données			
Tranche	*Temps Tot. (s)*	*Temps Com. (s)*	*Temps Trait. (s)*
1250	62,620	57,083	2,495
3125	29,893	24,081	2,403
6250	19,529	14,243	2,402
15625	14,510	8,556	2,364
31250	12,718	7,580	2,354
Récupération de 3 cases de données			
Tranche	*Temps Tot. (s)*	*Temps Com. (s)*	*Temps Trait. (s)*
1250	82,619	74,227	3,996
3125	38,645	30,824	3,815
6250	26,028	18,387	3,775
15625	19,638	11,436	3,756
31250	17,825	9,233	3,735

Table 7-12: Performances de récupération de *f* cases de données, *f* = 1, 2, 3, [Taille récupérée *f3,125MO ; TCP/IP ; décodage RS ; CG(2^8); SDDS2000 ; 1ére config.].**

Remarquons que :

* Les temps de traitement –essentiellement les temps de décodage, sont indépendants du paramètre *tranche* et sont linéaires au nombre de cases récupérées.
* Le temps de communication décroît quand la valeur du paramètre *tranche* augmente, et représente de 50 à 90% du temps total. Ceci est raisonnable, vu que le paramètre *tranche* détermine le nombre d'itérations de récupération, et par la suite le nombre de connexions TCP effectuées (voir analyse §6.5.5 du Scénario 6-16).
* Les meilleurs résultats correspondent à la récupération par itération de la totalité des enregistrements des cases en échec, et où les meilleurs temps de communication sont réalisés.
* Les meilleures vitesses de récupération sont de 0,46MO/sec pour la récupération d'une case de données (3,125MO), de 0,49MO/sec pour la récupération de deux cases de données (6,250MO) et de 0,53MO/sec (9,375MO) pour la récupération de trois cases de données.
* Les résultats sont expectatifs, notons que la vitesse de récupération est plus performante plus le nombre *f* de cases de données en échec à récupérer augmente. Ceci est du à la factorisation des messages d'interrogation des cases disponibles. En effet, à chaque itération, nous interrogeons au maximum *m*-1 cases pour récupérer *f* **tranche* enregistrements.

7.9.2.3 Comparaison des Scénarios

Incontestablement, le scénario basé sur le protocole TCP/IP est plus fiable et performant que celui basé sur le protocole UDP. Le temps de récupération d'une case de données de 3,125MO UDP est de 17,375 sec et nous avons obtenu de meilleurs

résultats {13.65 sec, 8.292 sec, 6.7sec} moyennant le protocole TCP/IP avec des valeurs de *tranche* dans l'ensemble{6250, 15625, 31250}. D'ailleurs, la vitesse de récupération de trois cases de données par UDP est inférieure à celle de récupération d'une case d'une case de données par TCP.

7.9.3 Architecture de Cases SDDS-TCP

Dans ce qui suit, nous rapportons les performances de récupération de *f* cases de données dans chacune des configurations. Nous confrontons les résultats obtenus dans les deux configurations respectivement la première et la deuxième configuration, les deux algorithmes de décodage respectivement XOR et RS, et les deux corps de Galois respectivement CG(2^8) et CG(2^{16}).

Pour la deuxième configuration, nous rapportons aussi les performances du décodage RS optimisé : RS^+ (§4.3.3.2). Rappelons que le décodage RS^+ optimisé consiste à procéder au pré-calcul d'une part du *log* de la matrice *H* inversée et d'autre part du vecteur *b* en cas de récupération de plus qu'une case de données ($f > 1$).

7.9.3.1 Choix du Paramètre *Tranche*

Le choix de la valeur du paramètre *tranche* doit être judicieux, en plus de notre analyse du scénario en §6.5.5, dans ce qui suit, nous prouvons par expérimentation que des petites valeurs de *tranche* dégradent les performances.

Nous avons mené une série d'expérimentations dans la deuxième configuration consistant en la récupération d'une case de données de 3,125MO par décodage XOR dans CG(2^8).

Tranche	*Temps Tot. (ms)*	*Temps Com. (ms)*	*Temps Trait. (ms)*
1	165281	148961	1735
100	4172	3642	346
500	1641	1379	247

Table 7-13: Variation du temps de récupération en fonction du paramètre *Tranche*.

7.9.3.2 Récupération d'une Case de Données

Première Configuration

La Table 7-14 récapitule les performances de récupération d'une case de données obtenues dans la première configuration, dans les deux corps de Galois CG(2^8) et CG(2^{16}) et par les algorithmes de décodage respectivement XOR et RS.

Corps de Galois: CG(2^8)						
*	*	**XOR**		*	**RS**	*
* ***Tranche***	* *Tot. (s)*	*Trait. (s)*	*Com. (s)*	* *Tot. (s)*	*Trait. (s)*	*Com. (s)* *
1250	4,654	0,677	3,882	6,300	1,065	5,188
3125	5,027	0,701	4,288	4,638	1,066	4,027
6250	4,278	0,683	3,575	4,727	1,068	3,635
15625	2,498	0,696	1,790	3,155	1,053	2,117
31250	2,328	0,683	1,630	2,914	1,051	1,845

Corps de Galois: CG(2^{16})						
*	*	XOR		*	RS	*
* Tranche	*Tot. (s)	Trait. (s)	Com. (s)	*Tot. (s)	Trait. (s)	Com. (s) *
1250	5,410	0,521	4,438	5,500	0,766	4,705
3125	9,238	0,536	7,188	5,626	0,763	4,850
6250	4,524	0,536	3,976	6,606	0,750	5,831
15625	2,889	0,530	2,351	2,701	0,746	1,953
31250	2,591	0,536	2,048	2,111	0,751	1,355

Table 7-14: Performances de récupération d'une case de données. [Taille récupérée : 3,125MO ; TCP/IP ; SDDS-TCP; 1ére config.].

Le choix du corps de Galois et l'algorithme de décodage influent le temps de traitement. En effet, il y'a lieu de noter :

* Dans le corps de Galois CG(2^8), un décodage XOR améliore le temps de traitement de 35%, par rapport à un décodage RS.
* Dans le corps de Galois CG(2^{16}), un décodage XOR améliore le temps de traitement de 30%, par rapport à un décodage RS.
* Un décodage XOR dans le corps de Galois CG(2^{16}) améliore les temps de traitement de 23%, par rapport à un décodage XOR dans le corps de Galois CG(2^8).
* Un décodage RS dans le corps de Galois CG(2^{16}) améliore les temps de traitement de 29%, par rapport à un décodage RS dans le corps de Galois CG(2^8).

Le temps de communication décroît quand la valeur du paramètre *tranche* augmente, et compte de 70 à 83% du temps total. Ceci est raisonnable, vu que le paramètre *tranche* détermine le nombre d'itérations de récupération, et par la suite le nombre de connexions TCP effectuées (voir analyse scénario §6.5.5).

Lors des expérimentations, nous avons noté une variation des temps de communication. A titre d'exemple pour *tranche* égale à 1250, le temps minimum de communication obtenu est 1,631sec et le temps maximum est 5,376sec. La variation décroît pour des tranches plus importantes en taille. Les temps de communication indiqués sont des moyennes des temps obtenus en écartant les valeurs extrêmes.

Deuxième Configuration

La Table 7-15 récapitule les performances de récupération d'une case de données obtenues dans la deuxième configuration, dans les deux corps de Galois CG(2^8) et CG(2^{16}) et par les algorithmes de décodage XOR, RS et RS$^+$.

Corps de Galois: CG(2^8)									
*Tranche	*	XOR		*	RS		*	RS$^+$	
*	* Tot.(s)	Trait.(s)	Com.(s)	* Tot.(s)	Trait.(s)	Com.(s)	* Tot.(s)	Trait.(s)	Com.(s)
1250	0,883	0,332	0,474	1,130	0,673	0,420	1,000	0,630	0,354
3125	0,677	0,255	0,364	1,125	0,692	0,359	0,943	0,626	0,307
6250	0,646	0,245	0,349	1,083	0,672	0,376	0,932	0,630	0,297
15625	0,760	0,245	0,433	1,209	0,683	0,442	0,932	0,626	0,301
31250	0,739	0,245	0,448	1,151	0,677	0,422	0,953	0,625	0,328

Corps de Galois: CG(2^{16})									
*Tranche	*	XOR		*	RS		*	RS^+	
*	* *Tot.(s)*	*Trait.(s)*	*Com.(s)*	* *Tot.(s)*	*Trait.(s)*	*Com.(s)*	* *Tot.(s)*	*Trait.(s)*	*Com.(s)*
1250	0,625	0,266	0,348	0,734	0,349	0,365	0,734	0,349	0,365
3125	0,588	0,255	0,323	0,688	0,359	0,323	0,688	0,359	0,323
6250	0,552	0,240	0,312	0,656	0,354	0,297	0,656	0,354	0,297
15625	0,562	0,255	0,302	0,667	0,360	0,297	0,667	0,360	0,297
31250	0,578	0,250	0,328	0,688	0,360	0,328	0,688	0,360	0,328

Table 7-15: Récupération d'1 case de données –[Taille récupérée 3,125MO ; TCP ; SDDS-TCP; 2éme config.].

Le choix du corps de Galois et l'algorithme de décodage influent sur le temps de traitement. En effet, il y a lieu de noter :

* Dans le corps de Galois CG(2^8), un décodage XOR améliore le temps de traitement de 61%, par rapport à un décodage RS.
* Dans le corps de Galois CG(2^{16}), un décodage XOR améliore le temps de traitement de 30%, par rapport à un décodage RS.
* Un décodage XOR dans le corps de Galois CG(2^{16}) dégrade les temps de traitement de par rapport à un décodage XOR dans le corps de Galois CG(2^8).
* Un décodage RS dans le corps de Galois CG(2^{16}) améliore les temps de traitement de 44%, par rapport à un décodage RS dans le corps de Galois CG(2^8).

Comparaison des Résultats des 2 Configurations

Nous comparons les résultats obtenus pour chacune des configurations, en prenant en considération, primo la composante temps de traitement dépendant de la vitesse du processeur, et secundo la composante temps de communication dépendant du réseau. Par rapport à la première configuration, la deuxième configuration améliore le temps de décodage XOR requis pour récupérer une case de données de 62% dans le corps CG(2^8) et de 50% dans le corps CG(2^{16}). En ce qui concerne, les temps de communication, la deuxième configuration l'améliore de 70-96% par rapport à la première configuration.

7.9.3.3 Récupération de 2 Cases de Données

Première Configuration

La Table 7-16 récapitule les performances de récupération d'une case de données obtenues dans la deuxième configuration, dans les deux corps de Galois CG(2^8) et CG(2^{16}) et par les algorithmes de décodage respectivement XOR et RS.

*	*	CG(2^8)		*	CG(2^{16})	*
* ***Tranche***	* *Tot. (s)*	*Trait. (s)*	*Com. (s)*	* *Tot. (s)*	*Trait. (s)*	*Com. (s)* *
1250	8,719	2,398	6,244	10,255	1,642	8,545
3125	6,664	2,402	7,217	7,191	1,647	7,096
6250	10,477	2,373	8,079	7,666	1,627	6,027
15625	8,536	2,368	5,372	6,369	1,630	4,732
31250	8,230	2,424	5,788	6,121	1,635	4,463

Table 7-16: Récupération de 2 cases de données -[Taille récupérée 6,250MO ; TCP ; décodage RS ; 1ére config.].

Lors des expérimentations, nous avons noté une variation des temps de communication. A titre d'exemple pour *tranche* égale à 1250, le temps minimum de communication obtenu est 4,507sec et le temps maximum est 6,759 sec. La variation décroît pour des tranches plus importantes en taille.

Le gain en performance réalisé en décodage de CG(2^{16}) par rapport à CG(2^8) est égal à 32%. Notons que nous avons obtenu une amélioration du même ordre dans le cas de récupération d'une case de données : 29% -cas de décodage RS.

Deuxième Configuration

La Table 7-17 récapitule les performances de récupération de deux cases de données, obtenues dans la deuxième configuration, dans les deux corps de Galois CG(2^8) et CG(2^{16}) et par les algorithmes de décodage RS et RS$^+$.

Corps de Galois: CG(2^8)						
*Tranche	*	RS		*	RS$^+$	*
*	* *Tot.(s)*	*Trait.(s)*	*Com.(s)*	* *Tot.(s)*	*Trait.(s)*	*Com.(s)**
1250	1,865	1,248	0,472	1,593	1,171	0,391
3125	1,791	1,235	0,364	1,508	1,172	0,329
6250	1,844	1,261	0,380	1,468	1,149	0,303
15625	1,781	1,255	0,401	1,464	1,156	0,302
31250	1,776	1,250	0,422	1,469	1,146	0,323

Corps de Galois: CG(2^{16})						
*Tranche	*	RS		*	RS$^+$	*
*	* *Tot.(s)*	*Trait.(s)*	*Com.(s)*	* *Tot.(s)*	*Trait.(s)*	*Com.(s)* *
1250	1,234	0,590	0,519	0,976	0,577	0,375
3125	1,172	0,599	0,400	0,932	0,589	0,338
6250	1,172	0,598	0,365	0,883	0,562	0,321
15625	1,146	0,609	0,443	0,875	0,562	0,281
31250	1,088	0,599	0,442	0,875	0,562	0,313

Table 7-17: Performances de récupération de 2 cases de données -[Taille récupérée 6,250MO ; TCP ; Architecture SDDS-TCP; 2éme config.].

Remarquons que :

* Par rapport à un décodage RS ou RS$^+$ dans CG(2^8), un gain en performance en moyenne de 52% est réalisé resp. en décodage RS ou RS$^+$ dans le corps CG(2^{16}). Nous avons obtenu une amélioration du même ordre d'échelle dans le cas de récupération d'une case de données : 44% -cas de décodage RS.

* Par rapport à un décodage RS, le décodage RS$^+$ améliore les performances de *Temps de Traitement* de 7,28% dans le corps de Galois CG(2^8) et de 4,77% dans CG(2^{16}). L'amélioration est plus importante que dans le cas de la récupération d'une case de données, car le pré-calcul du log affecte le vecteur *b* outre la matrice *H* inversée.

Comparaison des Résultats des 2 Configurations

La deuxième configuration améliore le temps de traitement contre la première configuration de 48% pour CG[2^8], et de 63% pour CG(2^{16}). En ce qui concerne, le temps de communication, la deuxième configuration l'améliore en moyenne de 83-96%.

7.9.3.4 Récupération de 3 Cases de Données

Première configuration

*	*	CG[2^8]		*	CG[2^{16}]		*
* ***Tranche***	* *Tot. (s)*	*Trait. (s)*	*Com. (s)*	* *Tot. (s)*	*Trait. (s)*	*Com. (s)*	*
1250	14,277	3,766	10,795	12,480	2,592	9,811	
3125	13,517	3,780	9,799	11,017	2,588	8,373	
6250	14,631	3,762	10,773	10,063	2,598	7,437	
15625	11,646	3,798	7,834	8,021	2,590	5,421	
31250	10,823	3,760	7,032	8,615	2,577	6,019	

Table 7-18: Performances de récupération de 3 cases de Données -[Taille récupérée 9,325MO ; TCP ; décodage RS ; SDDS-TCP; 1ére config.].

Lors des expérimentations, nous avons noté une variation des temps de communication. A titre d'exemple pour *tranche* égale à 1250, le temps minimum de communication obtenu est 6,29sec et le temps maximum est 13,810 sec. La variation décroît pour des tranches plus importantes en taille.

Le gain en performance de décodage, réalisé pour la récupération de trois cases de données, en utilisant CG(2^{16}) au lieu de CG(2^8) est de 31,38%. Notons que nous avons obtenu une amélioration du même ordre d'échelle dans le cas de récupération de deux cases de données : 32%, et le cas de récupération d'une case de données : 29% -cas de décodage RS.

Deuxième Configuration

La Table 7-19 récapitule les performances de récupération de trois cases de données obtenues dans la deuxième configuration, dans les deux corps de Galois CG(2^8) et CG(2^{16}) et par les algorithmes de décodage RS et RS$^+$.

Corps de Galois: CG(2^8)

**Tranche*	*	RS		*	RS$^+$		*
*	* *Tot.(s)*	*Trait.(s)*	*Com.(s)*	* *Tot.(s)*	*Trait.(s)*	*Com.(s)**	
1250	2,457	1,874	0,459	2,187	1,749	0,407	
3125	2,495	1,864	0,407	2,133	1,743	0,353	
6250	2,448	1,875	0,380	2,094	1,711	0,374	
15625	2,443	1,860	0,442	2,124	1,742	0,359	
31250	2,464	1,865	0,459	2,101	1,734	0,367	

Corps de Galois: CG(2^{16})

**Tranche*	*	RS		*	RS$^+$		*
*	* *Tot.(s)*	*Trait.(s)*	*Com.(s)*	* *Tot.(s)*	*Trait.(s)*	*Com.(s)*	*
1250	1,589	0,922	0,522	1,281	0,828	0,406	
3125	1,599	0,928	0,383	1,250	0,828	0,390	
6250	1,541	0,907	0,401	1,211	0,852	0,352	
15625	1,578	0,891	0,520	1,188	0,823	0,361	
31250	1,468	0,906	0,495	1,203	0,828	0,375	

Table 7-19: Performances de récupération de 3 cases de Données -[Taille récupérée 9,325MO ; TCP ; SDDS-TCP; 2ére config.].

* Le gain en performance de décodage, réalisé pour la récupération de trois cases de données, en utilisant CG(2^{16}) au lieu de CG(2^8) et quelque soit l'algorithme de décodage RS ou RS$^+$, est de 51%. Notons que nous avons obtenu une amélioration du même ordre d'échelle dans le cas de récupération de deux cases de données :

52%, et le cas de récupération d'une case de données : 44% -cas de décodage RS et 0% -cas de décodage XOR.

* L'optimisation du décodage par le pré-calcul du *log* améliore les *Temps de Traitement* de 7,046% pour CG(2^8) et de 8,76% pour CG(2^{16}).

Comparaison des Résultats des 2 Configurations

La deuxième configuration améliore le temps de décodage contre la première configuration de 38% pour CG(2^8), et de 56% pour CG(2^{16}). Remarquons que nous avons obtenu des résultats similaires en comparant les performances de décodage pour la récupération de deux cases de données 48% et 63% et celles pour la récupération d'une case de données 62% et 50%.

La deuxième configuration améliore également le temps de communication au 90-96%.

7.9.3.5 Conclusion

Le temps de communication varie peu, récupérer une, deux ou trois cases de données. En effet, le scénario de récupération interroge dans tous les cas, m-1 cases disponibles, mais envoie f tampons par itération vers les cases de secours. Nous notons une linéarité dans les performances de décodage, entre récupérer une, deux ou trois cases de données.

Le temps de communication est le temps passé pour réceptionner l'ensemble des tampons de la part des cases disponibles, en plus du temps requis pour envoyer les tampons des enregistrements récupérés. Le temps d'envoi s'avère non coûteux, ceci est du à une optimisation introduite dans le scénario, qui parallélise l'interrogation des cases disponibles et l'envoi des tampons de récupération. En effet, une fois le décodage effectué, les requêtes d'interrogation des cases disponibles sont envoyées, puis les données récupérées sont envoyées aux cases de secours. Ce qui fait que la préparation des tampons par les cases participant à la récupération, et l'envoi des tampons de récupération vers les cases de secours, se font en même temps.

Remarquons également que les performances de récupération d'une case de données sont meilleures que celles de création d'une case de parité. La différence entre les deux scénarios est que le scénario de création d'une case de parité est pénalisé par les écritures, en effet la réception de chaque tampon est suivie de mise à jour du contenu de la case de parité.

7.10 Conclusion

Tout au long du présent chapitre, nous avons rapporté les résultats de séries d'expérimentations. Ces dernières ont été menées afin d'évaluer nos choix de conception architecturaux et algorithmiques de codage/ décodage.
Les résultats obtenus sont prometteurs dans le domaine, et prouvent la faisabilité d'une structure de données distribuée et scalable. Néanmoins, dans le but d'avoir de meilleures performances, des améliorations peuvent être apportées, et c'est ce que le chapitre suivant évoque.

CHAPITRE

8 CONCLUSION & TRAVAUX FUTURS

8.1 Introduction

Le sujet de recherche de cette thèse est au confluent de plusieurs domaines de recherche, notamment la conception de structures de données distribuées, les mécanismes de haute disponibilité et de la théorie de l'information. Dans ce qui suit, nous évoquons des perspectives de travaux futurs concernant une mise en œuvre plus performante du prototype, d'un nouveau schéma LH^*_{RS}, et l'évaluation de nouveaux mécanismes de haute disponibilité.

8.2 Mise en Œuvre plus Performante

Cette section évoque des améliorations de conception des scénarios en vue de réaliser de meilleures performances. Nous nous intéressons particulièrement aux scénarios de récupération de cases, de récupération d'enregistrements de données, de transfert de mises à jour vers les cases de parité, d'éclatement de cases de données et enfin à la structure de données des cases.

8.2.1 Récupération de Cases

Afin d'introduire plus de parallélisme dans le scénario de récupération, nous proposons les stratégies suivantes:

La première consiste à accélérer le temps de récupération au niveau du gestionnaire de récupération. Rappelons que la requête de récupération de cases est prise en charge par un seul *thread* de travail. Ce dernier interroge les cases participant à la récupération, pour qu'elles envoient leurs contenus. Une fois les tampons de récupération réceptionnés, il procède à la récupération de la *tranche* d'enregistrements à récupérer. Afin d'accélérer le processus de décodage, nous proposons que le *thread* de travail crée des *threads fils temporaires* ou charge le *pool de threads* de travail de récupérer une partie de la *tranche* d'enregistrements. Enfin, les tampons contenant des enregistrements récupérés peuvent être fusionnés et envoyés en une seule fois ou chaque *thread* prend en charge la procédure d'envoi.

La deuxième consiste en la désignation de plusieurs gestionnaire de récupération. Dans le scénario de récupération proposé, le gestionnaire de récupération interroge un ensemble de cases disponibles fixé par le coordinateur. Afin de réaliser un équilibre de charge entre les différentes cases disponibles et inférer un parallélisme au scénario de récupération, nous pourrions créer des groupes de cases disponibles et désigner plus qu'un gestionnaire de récupération. Tel que, chaque gestionnaire est responsable de la récupération d'une *tranche* d'enregistrements.

8.2.2 Récupération d'Enregistrements de Données

La case de parité désignée par le coordinateur pour récupérer un enregistrement de données de clef *C*, ne sachant pas son rang *r*, exécute un parcours séquentiel des enregistrements de parité dont le but d'identifier l'enregistrement de parité de clef *r* et tel que *C* appartient au champ *liste des clefs* de *r*.

Algorithme 8-1: Recherche d'un enregistrement de données de clef *C* dans une case de parité.

d étant le numéro logique de la case de données à laquelle C appartient,
m étant la taille d'un groupe de parité,
indice est l'indice de la case de données *d* dans son groupe de parité,
indice ← *d* modulo *m*
i ← 0
Tant que (Non Fin-Case) & (Non Trouvé)
    Analyser l'enregistrement de parité de clef *i*
    Si (*C* = *Ep.Liste des Clefs*[*indice*]) Alors
        Trouvé ←Vrai
        r ← *Ep*.Clef
    Fin Si
    i ← *i* + 1
Fin Tant que
Retourner *r*

Afin d'accélérer la recherche d'un enregistrement de données dans une case de parité, nous proposons l'implantation de *m* index [*clef enregistrement de données – clef enregistrement de parité*], qui accéléreront la détermination du rang d'un enregistrement de données sachant sa clef.

L'inconvénient majeur de l'indexation est la complexité de mise à jour des indexes. En effet, ces index sont à mettre à jour suite aux requêtes d'insertion et de suppression d'enregistrements de données, et à l'éclatement d'une case de données il serait même préférable de recalculer leurs contenus.

8.2.3 Transfert des Mises à Jour vers les Cases de Parité

Les mises à jour des cases de parité, à l'exception de celles dues au scénario d'éclatement, sont à présent acheminées moyennant le protocole UDP et une stratégie d'acquittement basée sur le principe de conservation de messages. Les messages, dont l'acquittement a

échoué, sont déplacés vers une liste de messages non acquittés. Il faudrait prévoir un traitement de cette liste.

8.2.4 Eclatement d'une Case de Données

Les expérimentations montrent une dégradation du temps de réponse suite à un éclatement d'une case de données. L'éclatement doit être transparent au client et sans incidence sur les performances. Nous proposons une stratégie basée sur les files pour absorber les requêtes du client. Nous proposons l'implantation de deux files, une pour les enregistrements à insérer en local et une pour les enregistrements partants vers la nouvelle case. A la fin du processus d'éclatement le contenu de chaque file est traité de la manière appropriée.

8.2.5 La Structure de Données d'une Case

Notons que toute opération de mise à jour sur la structure de données des cases verrouille la case pour l'exécution exclusive de cette opération. Vu que les cases ont une architecture multitâche, il est impératif de permettre l'exécution en parallèle de plusieurs opérations de mise à jour. Pour ce, il est nécessaire de fragmenter la structure de données d'une case en structures de données indépendantes, telle que chaque opération de mise à jour verrouille une seule partie de la case.

8.3 Perspectives d'un Nouveau Schéma LH*$_{RS}$

8.3.1 Le *coordinateur*

Jusqu'à présent le schéma LH*$_{RS}$ ne supporte pas l'échec du coordinateur. Un scénario de récupération du coordinateur est à mettre en œuvre. Ce scénario doit tenir compte du fait que le coordinateur centralise toutes informations concernant l'ordre des éclatements, l'augmentation de la haute disponibilité, les adresses des cases et les propriétés des groupes de parité.
La conception du coordinateur en elle-même est problématique. En effet, nous avons le choix de concevoir le coordinateur en tant qu'une entité à part entière dans notre système, ou en tant qu'une couche additionnelle qui s'ajoute à n'importe quelle case de données, voire même de décentraliser le coordinateur de façon à avoir un coordinateur par groupe de parité.

8.3.2 Schéma d'Allocation

Il s'agit de définir une stratégie d'allocation de cases aux sites. En effet, faute de serveurs un schéma d'allocation judicieux permettra d'allouer les cases de parité sur les nœuds du groupe suivant. En cas d'échec d'une case de données du groupe g, il est possible que nous soyons amenés à récupérer une case de parité du groupe g-1.
Le processus d'allocation peut se produire à l'inverse. En effet, les cases de parité d'un groupe g sont allouées, avant la création des cases de données du groupe g+1. Si suite à

un éclatement, un nouveau groupe *g* est initié, ses cases de données sont allouées sur les serveurs de parité du groupe *g*-1, et ses cases de parité sont allouées à de nouveaux nœuds, qui hébergeront plus tard les cases de données du groupe *g*+2.
Ce schéma d'allocation augmente la fiabilité des groupes, puisqu'une case de données et une case de parité du même groupe ne risquent pas de coexister sur le même nœud.
La stratégie d'allocation peut être à la charge de l'administrateur ou/et à défaut décidée par le coordinateur, dans le ce sens où ce dernier motive le choix d'un site par rapport à un autre en cas de création d'une nouvelle case. Notons qu'il serait utile de prévoir un utilitaire permettant le changement d'adresse d'une case et toute la procédure conséquente.

8.3.3 « Parity Declustering »[8]

Il s'agit de définir un schéma LH^*_{RS} implantant le *parity declustering* (dégroupage des données de parité). L'objectif de ce procédé est de distribuer les unités de parité sur l'ensemble des disques, afin d'éviter les goulots d'étranglement sur les disques de parité. La surcharge d'un disque de poarité peut aoparaitre suite aux misess derniers peuvent apparaître sont par des mises à jour se destinant au même disque de parité. Les techniques de *parity declustering* permettent la répartition de la charge des mises à jour sur *k* disques. Une solution à ce problème est expliquée dans [LMS04].

8.4 Expérimentation de Techniques Alternatives de Codage/Décodage

La recherche d'un code moins complexe. En effet, nous estimons que le chapitre consacré à l'état de l'art a cité à titre non exhaustif les travaux de recherche conduits dans le domaine. L'implantation de ces algorithmes de codage et de décodage, de façon indépendante du prototype actuel, dans le but de les comparer sur le plan expérimental est requise.

8.5 Communication plus Rapide

8.5.1 Communication par UDP

Certains messages sont plus prioritaires que d'autres, et ne réquisitionnent pas les mêmes ressources partagées, notamment les messages de service. Nous pourrions concevoir un port d'écoute UDP réservé à un service déterminé pour chaque serveur.

---

[8] Nous avons utilisé le terme en anglais, car nous n'avons pas connaissance du terme français équivalent.

8.5.2 I/O Completion Ports

Les récents travaux de l'équipe de Douglas C. Schmidt [HPS97, HPS00], proposant l'implantation des APIs de communication non bloquantes offertes de l'environnement Windows, révèlent l'efficacité de APIs telles que : *TransmitFile* (plus adaptée au transfert de fichiers volumineux sous charge), les *Overlapped I/Os* et les *I/O Completion ports* [JD00]. Ces techniques exploitent des stratégies de 'Event dispatching' asynchrones offertes dans les environnements Windows, où les opérations asynchrones sont prises en charge par le système d'exploitation, telle que l'application n'est notifiée qu'à la fin de la l'achèvement de l'appel qualifié de non bloquant. Le cadre de l'expérimentation est le serveur web JAWS, et une comparaison est faite avec les performances de la première architecture implantée en utilisant les BSD *sockets* et la librairie de *threads* POSIX.

8.5.3 Le Protocole T/TCP

Le protocole TCP repose sur un mode connecté, alors que le protocole UDP fonctionne en mode non connecté. Les particularités de ces protocoles les rendent utiles dans différents domaines. Du fait de son fonctionnement en mode non connecté, UDP suppose toujours que l'hôte destinataire a reçu les données correctement. Même si UDP n'est pas fiable, il est plutôt rapide et pratique pour les applications où on préfère la vitesse à la fiabilité. D'un autre côté, TCP est intrinsèquement fiable mais plus lent qu'UDP.

Beaucoup d'applications sont basées sur la transaction plutôt que sur la connexion, mais sont néanmoins dépendantes de TCP et du surcoût associé. L'autre solution est UDP, mais ce protocole n'intégrant pas les mécanismes d'expiration et de retransmission, il incombera aux développeurs l'implantation du contrôle de flux, la vérification de l'intégrité des données transmise, le traitement des dédoublements de paquets, l'ordonnancement des messages etc.

Chacun des deux protocoles se trouve à un bout de l'échelle : TCP possède la fiabilité au dépend de la vitesse et c'est le contraire pour UDP. T/TCP est une extension expérimentale du protocole TCP et se présente par son concepteur Bob Braden le successeur aussi bien de TCP que de UDP pour certaines applications. C'est un protocole transactionnel basé sur un minimum de transferts de segments, ainsi il n'est pas affecté par les problèmes de vitesse associés à TCP. Reposant sur ce dernier, il ne pâtit pas du manque de fiabilité de UDP. La RFC 1379 [B92] aborde les concepts qu'impliquait l'extension de TCP afin de permettre un service transactionnel.

Les principales modifications sur le protocole TCP évoqués dans la RFC 1379 sont l'escamotage de la négociation tripartite et le raccourcissement de l'état d'attente TIME-WAIT de 240 à 12 secondes. La spécification complète de T/TCP se trouve dans la RFC 1644 [B94].

Des extensions, implantations et évaluations de T/TCP ont été faites. Dans la matière, il y'a lieu de citer Richard W. Stevens [S96], Mark Stacey et al. [S98] [SNG99], Ren Bin & Zhang Xioalan [BX00] et Michael Mansberg [Ma02].

Etant donné que notre prototype est implanté dans l'environnement *Windows* de Microsoft, et que nous n'avons pas accès au code des APIs de la bibliothèque *WinSockets*. Nous n'avons pas pu tester les performances de T/TCP.

8.6 Conclusion

Pour conclure, ce manuscrit résumant notre travail de quatre ans durant, qui nous a certes enrichis. Nous espérons qu'il sera utile à la communauté scientifique.

BIBLIOGRAPHIE

[A01] B. Aboba, *Pros and Cons of Upper Layer Network Access*, March 2001. http://www.drizzle.com/~aboba/IEEE, http://www.ietf.org/proceedings/01mar/slides/burp-1/

[ABC97] G. A. Alvarez, W. A. Burkhard, F. Cristian, *Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering*, Proc. of the 24th annual intl. Symp. on Computer architecture p.62-72, 1997.

[AFS] Open AFS Home page : http://www.openafs.org/.

[B00] F. S. Bennour, *Contribution à la Gestion de Structures de Données Distribuées et Scalables,* Thèse de doctorat, Juin 2000, Université Paris Dauphine.

[B01] B. Beej, *Beej's Guide to Network Programming Using Internet Sockets*, http://www.ecst.csuchico.edu/~beej/guide/net/html/.

[B02] F. S. Bennour, *Performance of The SDDS LH*LH under SDDS-2000*, Proc. of WDAS 2002, p.1-14 Carleton scientific.

[B3M95] Blaum, M., Brady, J., Bruck, J., Menon, J., *EVENODD: An efficient Scheme for Tolerating Double Disk Failures in RAID Architectures*. IEEE Trans. Computers, vol. 44, p. 192-202, 1995.

[B92] B.Braden, *Extending TCP for Transactions –Concepts*, RFC1379, 1992. http://www.faqs.org/rfcs/rfc1379.html.

[B94] B. Braden, *T/TCP –TCP Extensions for Transactions Functional Specification*, RFC1644, 1994.

[BDET00] W.J. Bolosky, J.R. Douceur, D. Ely & M. Theimer, Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs, SIGMETRICS 2000, p.34-43.

[BDNL00] F. S. Bennour, A. W. Diéne, Y. Ndiaye, W. Litwin, *Scalable and Distributed Linear Hashing* $LH*_{LH}$ *under Windows NT*, SCI-2000.

[BKL+95] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby & D. Zuckerman, *An XOR-Based Erasure-Resilient Coding Scheme*, ICSI Tech. Rep. TR-95-048, 1995.

[BKMM98] K. Berket, R. Koch, L.E. Moser, P.M. Melliar-Smith, *Timestamp Acknowledgments for Determining Stability*, Proc. of the 2nd Inter. Conf. On Parallel and Distributed Computing and Network, 1998. http://george.lbl.gov/InterGroup/papers/pdcn98/pdcn98.pdf.

[BM00] P. Bozanis, Y. Manolopoulos, DSL : Accomodating Skip Lists in the SDDS Model, Proc. of WDAS 2000, p.1-9, Carleton scientific.

[BM02] P. Bozanis, Y. Manolopoulos, *LDT: A Logarithmic Distributed Search Tree*, Proc. of WDAS 2002, p.121-132, Carleton scientific.

[BM93] W. A. Burkhard, J. Menon, *Disk Array Storage System Reliability*, The 23rd Intl. Symposium on Fault-Tolerant Computing, Toulouse, 1993.

[BWWG02] M. Bakkaloglu, J. J. Wylie, C. Wang, G. R. Ganger, *On Correlated Failures in Survivable Storage Systems*, Rapport Technique CMU-CS-02-129, U. Carnegie Mellon.

[BX02] R. Bin & Z. Xioalan, *Enhanced Transaction TCP –Design and Implementation*, http://ttcplinux.sourceforge.net/theses/ETTCP.pdf http://ttcplinux.sourceforge.net/theses/TTCP_impl.pdf

[C02] R. J. Chevance http://rangiroa.essi.fr/cours/systeme-information/01-dimensionnement-systemes.pdf, http://rangiroa.essi.fr/cours/systeme-information/01-haute-disponibilite.pdf

[CEG+04] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, S. Sankar, *Row-Diagonal Parity for Double Disk Failure Correction*, Proc. of the 3rd USENIX –Conf. On File and Storage Technologies, Avril 2004.

[CMST03] J. A. Cooley, Jeremy L. Mineweaser, Leslie D. Servi & Eushiuan T. Tsung, *Software-based Erasure Codes for Scalable Distributed Storage*, 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03) April 07 - 10, 2003 San Diego, California.

[CPR] Contingency Planning Research, *Financial Impact of System Failure: Statistics on 1996 Downtime Cost*, http://www.contingencyplanningresearch.com/cod.htm

[CW98] A. Cohen, M. Wooding, *Win32 Multithreaded Programming*, O'REILLY, 1998.

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

[D01-p] A.W. Diène, *Prototype de la SDDS RP**, CERIA Lab., Université Paris Dauphine. http://ceria.dauphine.fr/aly/aly.html.

[D02] http://www.cs.buffalo.edu/faculty/bina/cse510/DistributedFileSystems.ppt

[D87] C. J. Date, *Twelve Rules for a distributed Database*, Computer World 2 (23) p.77-81 1987.

[D93] R. Devine, *Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm*, Proc. of the 4th Intl. Foundation of Data Organization and Algorithms –FODO, 1993.

[DA99] J. Durbin, L. Ashdown, *Oracle8i : Distributed Database Systems*, Oracle Press, 1999. http://sunsite.eunnet.net/documentation/oracle.8.0.4/server.804/a58247.pdf

[DG92] D. DeWitt & J. Gray, *Parallel Database Systems: The Future of High Performance Database Systems*, Communications of The ACM, June 1992, Vol.35 No.6, pp.85-97.

[DKW95] S. Dolev, M. Kate, J. L. Welch, A Competitive Analysis for Retransmission Timeout, 15th Inter. Conf on Distributed Computing Systems, ICDCS'1995. http://faculty.cs.tamu.edu/welch/papers/ntwks99.pdf.

[DKW95] S. Dolev, M. Kate, J. L. Welch, *A Competitive Analysis for Retransmission Timeout*, 15th Inter. Conf. On Distributed Computing Systems: ICDCS 1995. http://faculty.cs.tamu.edu/welch/papers/ntwks99.pdf.

[DL00] A. W. Diéne, W. Litwin, *Implementation and Performance Measurements of the RP* Scalable and Distributed Data Structure for Windows Multicomputers*, Intl. Workshop on Performance-Oriented Program Devpt for Distributed Architectures –PADDA 2001.

[DL01] A. W. Diéne, W. Litwin, Performance Measurements of RP*: *Scalable and Distributed Data Structure for Range Partitioning*, Intl. Conf. on Information Society in the 21st Century: Emerging Tech. And New Challenges, Japan 2000.

[F96] S. Frank, Une synthèse bibliographique sur les disques RAID, CNAM 1996. http://beru.univ-brest.fr/~singhoff/publications/ps/raid.ps

[FNPS79] R. Fagin, J. Nieverjelt, N. Pippenger, H. R. Strong, *Extendible Hashing – A Fast Access Method for Dynamic Files*, ACM Transactions on Database Sytems 1979.

[G03] A. Le Glaunec, *Corps de Galois : aide mémoire*, http://supelec-rennes.fr/ren/perso/aleglaun/corpsdegalois.pdf

[G94] S.Ghernaouti-Hélie, *Client/Serveur les outils du Traitement Réparti Coopératif*, Editions Masson 1994.

[G95] J. Gray, *Queues are Databases*, Proc. 7th High Performance Transaction Processing Workshop, Asilomar, CA, Sept 1995. http://www.research.microsoft.com/~Gray.

[G97] G. Gilder, *Fiber keeps its Promise: Get Ready Bandwidth will triple each year for the next 25*. Forbes, 7 April 1997. http://www.forbes.com/asap/97/0407/090.htm

[GG96] G. Gardarin, O. Gardarin, *Le Client-Serveur*, Editions Eyrolles 1996.

[GRS97] J. C. Gomez, V. Rego, V. S. Sunderam, *Efficient MultiThreaded User-Space Transport for Network Computing: Design and Test of the TRAP Protocol*, Journal of Parallel and Distrib. Comput. 40(1): 103-117 (1997).

[GS00] J. Gray, P. Shenoy, *Rules of Thumb in Data Engineering*, IEEE International Conference on Data Engineering, San Diego, April 2000.

[GS91] J. Gray, D. Siewiorek, *High Availability Computer Systems*, IEEE Computer, Vol.24, N°9, September 1991, pp. 39-48.

[GVAB96] J. Griffioen, R. Vingralek, T. A. Anderson, Y. Breitbart: *DERBY: A Memory Management System for Distributed Main Memory Databases*. RIDE-NDS 1996, pp. 150-159.

[HBC97] V. Hilford, F. B. Bastani, B. Cukic, *EH* - Extendible Hashing in a Distributed Environment*, COMPSAC '97 - 21st International Computer Software and Applications Conference.

[HGK+94] L. Hellerstein, G. Gibson, R.M. Karp, R.H. Katz, and D.A. Patterson, *Coding techniques for handling failures in large disk arrays*. Algorithmica, 12:182-208, 1994.

[HKM+88] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, *Scale and performance in a distributed file system*, ACM Transactions on Computer Systems, 6(1):51-81, February 1988.

[HO95] J. H.Hartman, John K. Ousterhout, *The Zebra Striped Network File System*, ACM Transactions on Computer Systems, Vol. 13, No. 3, Aug. 1995. http://www.cse.ucsc.edu/~sbrandt/courses/Fall01/290S/zebra.pdf.

[HPS00] J. Hu, I. Pyarali, D. C. Schmidt, *The Object-Oriented Design and Performance of JAWS: A High-performance Web Server Optimized for High-speed Networks*, The Parallel and Distributed Computing Practices journal, special issue on Distributed Object-Oriented Systems, Vol. 3, No. 1, March 2000. http://www.cs.wustl.edu/~schmidt/PDF/PDCP.pdf

[HPS97] J. Hu, I. Pyarali, D. C. Schmidt, *Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks*, Proceedings of the 2nd Global Internet Conference, Phoenix, Nov. 1997.

[IMT03] M. Isard, M. Manasse, C. Thekkath, *Koh-i-Noor Project*, http://research.microsoft.com/research/sv/kohinoor/

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

[JD00] A. Jones, A. Deshpande, *Windows Sockets 2.0: Write Scalable Winsock Apps Using Completion Ports*, MSDN Magazine: Oct. 2000, http://msdn.microsoft.com/msdnmag/issues/1000/Winsock/

[JK88] V. Jacobson, M. J. Karels, *Congestion Avoidance and Control*, Computer Communication Review, Vol. 18, No 4, pp. 314-329.

[KLR96] J. Karlson, W. Litwin & T. Risch, LH*LH: *A Scalable high performance data structure for switched multicomputers*, EDBT 96, Springer Verlag.

[KS92] J. Kistler and M. Satyanarayanan, *Disconnected Operation in the Coda File System*, ACM Transactions on Computer Systems, Vol. 10, No. 1, February 1992, pp. 3-25. http://www.cse.ucsc.edu/~sbrandt/courses/Fall01/290S/coda.pdf.

[KW94] B. Kroll, P. Vidmayer, *Distributing a Search Tree among a Growing Number of Processors*, ACM Intl. Conf. on Management of Data -SIGMOD 1994.

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

[L78] P. Larson, *Dynamic Hashing*, BIT Vol. 18-(2), 1978.

[L80] W. Litwin, Linear Hashing: A New Tool for File and Table Addressing, reprint from VLDB80 in Readings in Databases, M. Stonebraker, 2nd Edition, Morgan Kaufmann Publishers, 1994.

[L95] E. K. Lee, *Highly-Available, Scalable Network Storage*, Digest of Papers COMPCON 1995, strony 397-402. IEEE Computer Society Press, Marzec 1995.

[L97] R. Lindberg, *A JAVA Implementation of a Highly Available and Distributed Data Structure LH*g*, M.Sc. Thesis U. Linkoping, 1997. http://home.swipnet.se/~w-61244/ex-jobb.htm

[LC83] S. Lin & D.J. Costello, *Error Control Coding: Fundamentals and Application*, Prentice Hall, 1983.

[LF] http://www.linux-france.org/prj/jargonf/.

[LM+97] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, V. Steman, "*Practical Loss-Resilient Codes*", Proc. 29th Symp. on theory of computing, 1997,p.150-159.

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

[LMRS99] W. Litwin, J. Menon, T. Risch & J.E. Schwarz, *Design Issues for Scalable Availability LH* Schemes with Record Grouping*, DIMACS Workshop on distributed Data and structures, Carleton Scientific, 1999.

[LMS04] W. Litwin, R. Moussa, T. Schwarz, *Prototype Demonstration of* Enhanced *$LH*_{RS}$* architecture, VLDB 2004, Toronto Canada.

[LMS04-*b*] W. Litwin, R. Moussa, T. Schwarz, *$LH*_{RS}$ – A Highly-Available Scalable Distributed Data Structure*, soumis journal.

[LN+97] W. Litwin, A.M. Neimat, G. Levy, S. Ndiaye & T. Seck, *$LH*_{S}$: a high-availability and high-security Distributed Data Structure*, IEEE Workshop on Res. Issues in Data Engineering, 1997.

[LN96] W. Litwin & A.M. Neimat, *High Availability LH* Schemes with Mirroring*, Intl. Conf on Cooperating systems, Brussels, IEEE Press 1996.

[LNS94] W. Litwin, M.A. Neimat & D. Schneider, *RP*: A Family of Order-preserving Scalable Distributed Data Structures*, Proceedings of the 20th VLDB Conference, Satiago, Chili, 1994.

[LR97] W. Litwin & T. Risch, *LH*g: A High-Availability Scalable data Structure by Record Grouping*, Res. Rep. U. Paris9 & U. Linkoping, 1997.

[LS00] W. Litwin & T. Schwarz, *$LH*_{RS}$: A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes,* p.237-248, Proceedings of the ACM SIGMOD 2000.

[LT96] E. K. Lee and C. A. Thekkath. *Petal: Distributed virtual disks*. In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, *ASPLOS-VII,* pages 84-92. ACM, October 1996. http://www.thekkath.org/papers/petal.pdf.

[M00] R. Moussa, *Implantation partielle & Mesures de performances de $LH*_{RS}$*, Université Paris Dauphine, Mémoire de DEA, octobre 2000, http://ceria.dauphine.fr/Rim/dea.pdf.

[M02] R. Moussa, *Prototype Demonstration of $LH*_{RS}$*, Session Démo de WDAS 2002: Workshop on Distributed Data & Structures, Paris 2002.

[M03] G. E. Moore, *No Exponential is Forever … but We Can Delay 'Forever'*, International Solid State Circuits Conference, 2003. ftp://download.intel.com/research/silicon/Gordon_Moore_ISSCC_021003.pdf

[M65] G. E. Moore, *Cramming More Components Onto Integrated Circuits*, Electronics, Vol. 38, Num. 8, April 1965. ftp://download.intel.com/research/silicon/moorespaper.pdf.

[Ma02] M. Mansberg, *TCP/IP for Transactions*, http://www.embedded.com/story/OEG20020627S0027.

[MB00] D. MacDonal, W. Barkley, *MS Windows 2000 TCP/IP Implementation Details*, http://secinf.net/info/nt/2000ip/tcpipimp.html.

[MK96] Mogi & M. Kitsuregawa, *Hot Mirroring: A method for hiding parity update penality and degradation during rebuilds for RAID*, Proc. of ACM SIGMOD Conf, pp.183-194, June 1996.

[ML02] R. Moussa & W. Litwin, *Experimental Performance Analysis of LH*$_{RS}$ Parity Management*, Proc. In Informatics Carleton Scientific: Distributed Data & Structures 2002, p.87-98.

[MS04] R. Moussa, T. Schwarz, *Design and Implementation of LH*$_{RS}$ – A Highly-Available Scalable Distributed Data Structure*, WDAS 2004, Lausanne.

[MS77] F. J. MacWilliams & N. J. A. Sloane, *The theory of Error Correcting Codes*, North-Holland, New York, 1977.

[MTS04] M. S. Manasse, C. Thekkath, A. Silverberg, *A Reed-Solomon code for disk storage, and efficient recovery computations for erasure-coded disk storage*, WDAS Lausanne 2004.

[MW02] R. Moussa, W. Litwin, *Experimental Performance Analysis of LH*$_{RS}$ Parity Management*, Carleton Scientific Records of the 4th International Workshop on Distributed Data & Structure : WDAS 2002, p.87-97.

[NZT96] M.G. Norman, T. Zurek & P. Thanisch, *Much Ado About Shared-Nothing*, SIGMOD Record Vol 25 N°3, p.16-21.

[ÖV99] M.T. Özsu, P. Valduriez, *Principles of Distributed Database Systems*, Prentice Hall, 1999.

[P00] O. Pothier, *Les codes en bloc linéaires*, http://comelec.enst.fr/~pothier/poly/cours2/bloc.doc

[P00] V. Y. Pan, *Matrix Structure, Polynomial Arithmetic, and Erasure-Resilient Encoding/Decoding*, Proc. of the 2000 International Symposium on symbolic and algebraic computation, p.266-271.

[P04] J.F. Pillou, *Le client Serveur*, http://www.commentcamarche.net/cs/csintro.php3.

[P89] F. P. Preparata. *Holographic dispersal and recovery of information. IEEE Transactions on Information Theory*, 35(5):1123–1124, September 1989.

[P97] J. S. Plank, *A Tutorial on Reed-Solomon Coding for fault-Tolerance in RAID-like Systems*, Software– Practise & Experience, 27(9), Sept. 1997, pp 995- 1012,

[PB01] D. Patterson and A. Brown, *Recovery-Oriented Computing*, http://roc.CS.Berkeley.EDU/ , October 2001.

[PGK88] D. A. Patterson, G. Gibson & R. H. Katz, *A Case for Redundant Arrays of Inexpensive Disks*, Proc. of ACM SIGMOD Conf, pp.109-106, June 1988.

[PN01] A. di Pasquale, E. Nardelli, *A Very Efficient Order Preserving Scalable Distributed Data Structure*, Proc. of DEXA 2001, p.186-199, Springer Verlag.

[PN99] A. di Pasquale, E. Nardelli, *Balanced and Distributed Search Trees*, Proc. of WDAS 1999, Carleton scientific.

[R89] M. O. Rabin, *Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance*, Journal of ACM, Vol. 26, N° 2, April 1989, pp. 335-348.

[R96] L. Rizzo, *On the Feasability of Software FEC*, DEIT Tech. Report LR970131, 1996, http://www.iet.unipi.it/~luigi/softfec.ps.

[R97] L. Rizzo, *Effective Erasure Codes for reliable computer communication protocols*, ACM Computer Communication Review, Vol.27, Apr. 97, pp. 24-36.

[R98] D. Rubenstein, *Increasing the Functionality and Availability of Reed-Solomon FEC Codes: a Performance Study*, UMass CMPSCI Technical Report 98-31, August, 1998. http://www-net.cs.umass.edu/~drubenst/publish/publish.html

[RO91] M. Rosenblum, J. Ousterhout, *The Design and Implementation of a Log-Structured File System*, Proc. of the 13th Symposium on Operating Systems principles, 1991.

[RS60] I. Reed & G. Solomon, *Polynomial codes over certain Finite Fields*, Journal of the society for industrial and applied mathematics, 1960.

[S01] http://setiathome.ssl.berkeley.edu

[S02] S. Surana, « *Tornado Codes* », Scribe Notes for 15-853. http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/pscicoguyb/realworld/www/errorcorrecting.html.

[S02] T. J.E. Schwarz S.J., *Generalized Reed Solomon Codes for Erasure Correction in SDDS*, Carleton Scientific Records of the 4th International Workshop on Distributed Data & Structure: WDAS 2002, p.75-86.

[S79] A. Shamir, *How to Share a Secret*, Comm. ACM, Nov. 1979, p.612-613.

[S96] A. K. Sinha, *Network Programming in Windows NT*, Addison-Wesley Publishing Company, 1996.

[S96] R. W. Stevens, *TCP/IP Illustrated volume 3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols*, Addison-Wesley, 1996.

[S98] M. Stacey, *Code source de T/TCP -environnement Linux*, http://www.csn.ul.ie/~heathclf/fyp/.

[SB92] T. J. E. Schwarz, W. A. Burkhard, *RAID Organization and Performance*, Proc. Of the 12th Intl. Conf. on Distributed Computing Systems, Yokohama, 1992.

[SB95] T. J. E. Schwarz, W. A. Burkhard, *Reliability and Performance of RAIDs*, IEEE Transactions on Magnetics (T-MAG*)*, vol. 31, p. 1161-1166, March 1995.

[SGL+85] R. Sandberg, D. Goldberg, S. Leiman, D. Walch, B. Lyon, *Designing and Implementation of the Sun Network File System*, Proc. Of the USENIX 1985 Summer Conference, El Cerrito, CA, USA, June 1985.

[SM99] A. De Santis, B. Masucci, *Multiple Ramp Schemes*, IEEE Trans. Information Theory, Juillet 1999, p.1720-1728.

[SNG99] M. Stacey, J. Nelson, I. Griffin, *T/TCP: TCP for Transactions*, http://www.linuxgazette.com/issue47/stacey.html, traduit en français sur http://ftp.traduc.org/doc-vf/gazette-linux/html/1999/lecture/issue-46-47/lg46_47-fr-4.html#ss4.1.

[SS02] E. J. Schwabe, I. M. Sutherland, *Efficient Data Mappings for Parity-Declustered Data Layouts,* Juillet 2002. http://facweb.cs.depaul.edu/eschwabe/

[SS96] E. J. Schwabe, I. M. Sutherland, *Flexible Usage of Redundancy in Disk Arrays*, Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures, p.99-108, 1996. http://facweb.cs.depaul.edu/eschwabe/

[SSW02] K. Schlude, E. S. Soininen & P. Widmayer, *«Distributed Highly Available Search Trees»*, SIROCCO 2002, Proceedings in Informatics, p259-274.

[SW98] D. Schischkin, G. Weber, *The UniServer Architecture and its use for database Access*, *Pro.International Conference on Parallel and Distributed Processing Techniques and Applications*, p. 1397-1403, 1998. http://www.inf.fu-berlin.de/inst/ag-db/

[SW98] D. Schischkin, G. Weber, *The UniServer Architecture and its Use for Database Access*, Proc. International Conf. on Parallel and Distributed Processing Techniques and Applications, 1998.

[VBW94] R.Vingralek, Y.Breitbart & G.Weikum, *Distributed File Organisation with Scalable Cost/Performance*, ACM SIGMOD Int. Conf. on Management of Data, 1994.

[VBW98] R.Vingralek, Y. Breitbart & G. Weikum, *SNOWBALL : Scalable Storage on Networks of Worstations with Balanced Load*, Distributed and Parallel Databases 6(2) 1998, pp. 117-156. http://www.bell-labs.com/user/rvingral/publications.html

[VK01] S. Vaghani, P.Kantawala, *Simba: A Distributed Video File Server*, http://www-db.stanford.edu/~svaghani/projects/simba/simba-report.pdf

[W91] P. E. White, *RAID X Tackles Design Problems with Existing RAID Schemes*, ECC Technologies, Inc., 1991. ftp://members.aol.com/mnecctek/ctr1991.pdf

[WB94] S. B. Wicker & V. K. Bhargava, *Reed-Solomon codes and their applications*, IEEE Press, New York, 1994.

[WBP+01] J. J. Wylie, M. Bakkaloglu, V. Pandurangou, M. W. Bigrigg, S. Oguz, K. Tew, C. Williams, G. R. Ganger, P. K. Khosla, *Selecting the Right Data Distribution Scheme for a Survivable Storage System*, Rapport Technique CMU-CS-01-120, U. Carnegie Mellon.

[WBS+00] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliçcote, P. K. Khosla, *Survivable Storage Systems*, Proc. of IEEE 2000, p.61-68.

[WBV+96] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle, *A fast software implementation for arithmetic operations in GF(2^n)*, Advances in Cryptology, Proceedings Asiacrypt'96, LNCS 1163, Springer-Verlag, 1996, pp. 65-76. http://www.esat.kuleuven.ac.be/~dewin/

[WGBC00] M. Welch, S. D. Gribble, E. A. Brewer, D. Culler, *A Design Framework for Highly Concurrent Systems*, UC Berkeley Tech. Report UCB/CSD-00-1108, Avril 2000. http://www.eecs.harvard.edu/~mdw/papers/events.pdf.

[WK02] H. Weatherspoon & J. D. Kubiatowicz, *Erasure Coding vs. Replication: A quantitative Comparison*, Proceedings of the 1st International Workshop on Peer-to-Peer Systems, March 2002, p.328-338.

[X98] L. Xu, *Highly Available Distributed Storage Systems*, Ph.D. Thesis, California Institute of Technology, 1998. http://paradise.caltech.edu/~lihao/thesis.html

[XB99-a] L. Xu & Jehoshua Bruck, *X-Code: MDS Array Codes with Optimal Encoding*, IEEE Trans. on Information Theory, 45(1), p.272-276, 1999.

[XB99-b] L. Xu & J. Bruck, *Highly Available Distributed Storage Systems*, Proceedings of Workshop on Distributed High Performance Computing Lecture notes in Control & Information sciences, Springer Verlag, 1999.

[XBBW98] L. Xu, V. Bohossian, J. Bruck & D. G. Wagner, *Low Density MDS Codes and Factors of Complete Graphs*, Proc. of IEEE Symposium on Information Theory, 1998. http://www.cs.utk.edu/~plank/plank/papers/cs.96.332.html

ANNEXE A: DESCRIPTION DU PROTOTYPE

Dans le cadre de cette thèse, nous avons repris les programmes développés dans le cadre de mon mémoire de DEA [M00]. Ces derniers se basent sur une version du prototype LH^*_{RS} développé par M. Ljungtrom [L00], ajoutant lui-même une couche de fonctionnalités par rapport au prototype du gestionnaire LH^*_{LH} développé par F. Bennour [B00].

La maintenance du prototype était difficile. En effet, il incorporait des fonctions et variables propres à RP* [D01] sans aucune importance pour LH^*_{RS}, et toutes les fonctionnalités LH* et LH^*_{RS} étaient regroupées dans quatre fichiers. Cette difficulté a été également soulignée dans le rapport de mastère de M. Ljungtrom [L00].

Pour une meilleure fiabilité, un de nos premiers objectifs était de ré-organiser le prototype. Pour ce,

- Nous avons supprimé les fonctions et variables qui ne servaient pas, et ré-écrit des procédures pour soit répondre à nos besoins ou pour de meilleures performances.
- Nous avons ré-organisé les fichiers de façon à ce que le nom du fichier renseigne sur les procédures qu'il contienne. Ainsi, le fichier *insert.c* contient les procédures/ fonctions d'insertion, de même le fichier *split server.c* contient les procédures/ fonctions d'éclatement réseau niveau de la case en éclatement, etc.
- Initialement, les identificateurs de messages étaient des chaînes de trois caractères. Nous avons implanté les identificateurs de messages sous forme de directives, étant surtout plus significatifs et pratiques au vu du nombre croisant de messages. Ainsi, le fichier *IdMsge.c* énumère tous les identificateurs de messages en les classant en trois catégories : messages UDP, messages TCP et messages multicast.
- Nous avons implanté une nouvelle structure pour les cases de données et de parité pour mieux pointer la structure des enregistrements de données et de parité.

L'espace de travail du prototype comprend cinq projets que sont : le *client*, le *coordinateur*, le *serveur de données*, le *serveur de parité* et enfin la *table d'adresses*. Plusieurs versions ont été développées en utilisant des directives au niveau des projets. Ces directives permettent de rétablir une version bien déterminée en vue de refaire les expérimentations par exemple. Nous citons à titre non exhaustif les directives suivantes :

- La directive *NewMatrixVal* définie au niveau des projets serveur de données et client, et les directives *NewMatrix* et *OPTLOG*, définies au niveau du projet serveur de parité, une fois activées font marcher le prototype en mode calcul de parité optimisé.
- La directive *Multicast* définie au niveau de tous les projets, constitue la version du prototype, où désormais la table d'adresses statique a été remplacée par des structures

LH*$_{RS}$: A Highly Available Distributed Data Storage

Witold Litwin Rim Moussa

CERIA Lab., Université Paris Dauphine
FRANCE
Witold.Litwin@dauphine.fr Rim.Moussa@dauphine.fr

Thomas J.E. Schwarz, S.J.

Santa Clara University
USA
TSchwarz@scu.edu

Abstract

The ideal storage system is always available and incrementally expandable. Existing storage systems fall far from this ideal. Affordable computers and high-speed networks allow us to investigate storage architectures closer to the ideal. Our demo, present a prototype implementation of LH*$_{RS}$: a highly available scalable and distributed data structure.

1. Introduction

Scalable and Distributed Data Structures [SDDS] are intended for computers over fast networks, usually local networks, i.e. for the *multicomputers.* This new hardware architecture is promising and gaining in popularity. In spite of the advantages given by distributing data, vulnerability to failures remains a problem that grows with the number of machines supporting the SDDS.

Many approaches to build highly available, i.e., fault tolerating, distributed data storage systems have been proposed. They generally use either (*i*) data mirroring or (*ii*) parity calculus [WK02]. The latter approach uses erasure-correcting codes. The simplest codes, e.g. in RAID systems [PGK88], use XOR calculus for the tolerance of a single site failure. Multiple failures need more complex codes. These can be the binary codes [H94] for double or triple failure, or character codes, more generally. Examples of character codes are array codes such as the EVENODD code [BB94], the X-code [XB99] or Reed Solomon codes. The latter appear at present to be the best to deal with multiple failures [R89] [BK95][P97] [LS00] [ML02] [S02] [MS04] [LMS04] [M04].

Below, Section 2 recalls the LH*$_{RS}$ file structure. Section 3 overviews our bucket architecture. Section 4 presents the demonstration outline. Finally, performance results are given in section 5.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

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

2. LH*$_{RS}$ Scheme

LH*$_{RS}$ scheme is described with details in [LS00] and [LMS04]. An LH*$_{RS}$ file is subdivided into groups. Each group is composed of m Data Buckets and k Parity Buckets. Buckets are basically in distributed RAM, each at a different server node. The data buckets store the data records of the group. These are encoded into the parity records for high availability as follows in the parity buckets. Every data record has a rank r in its data bucket. It receives this rank upon insertion.

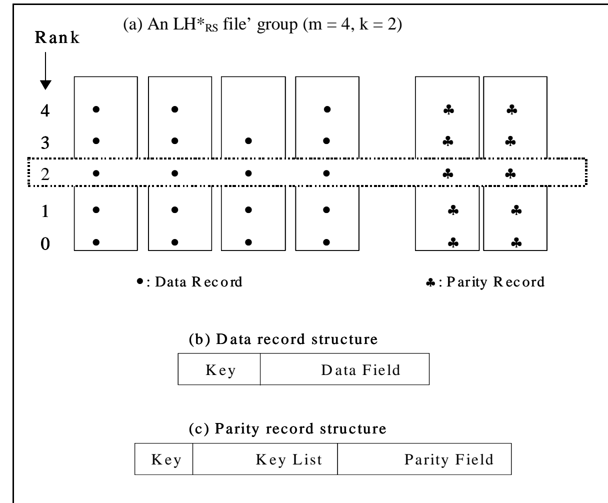


Figure 1: LH*$_{RS}$ file Structure

A *record group* consists of all records with the same rank in a *bucket group*. We construct parity records from data records having the same rank within data buckets forming a bucket group (Fig. 1(a)). The record grouping has an impact on the data structure of a parity record. Fig. 1(b-c) shows the structure of a data record and a parity record. The *key* field of the parity record is its rank; the *key list* keeps track of the data records in the record group. The parity field contains the actual parity symbols for the

record group that we calculate using our version of Reed Solomon codes.

Thanks to the coding scheme, we can calculate the contents of all records in a record group if we have access to m out of the $n = m+k$ records. This provides us with k availability. The actually parity calculation proceeds as follows: When a record is updated (and here an insert counts as a modification of a zero record) we calculate its Δ-record, which is the XOR of the old and the new data field. We send this Δ-record to all parity buckets, who update their parity field by XORing the Δ-record multiplied symbol-wise by a given fixed element, contained in a *parity matrix* **P**. The multiplication is done in a Galois field. According to our experiments we achieve the overall best performance with the Galois field with 2^{16} elements. Since we use the logarithmic method to multiply, we actually store directly the logarithms of **P** in a matrix **Q**. Thanks to an optimization of **P**, the first row and the first column of **P** contain only 1's coefficients. In this case, we obviously can do away with the multiplication. In any case, our experiments show that the processing overhead of Reed-Solomon is small. Table 1 below shows our demo matrices **P** and **Q**.

(P)			
	0001	0001	0001
	0001	eb9b	2284
	0001	2284	9e74
	0001	9e44	d7f1

(Q)			
	0000	0000	0000
	0000	5ab5	e267
	0000	e267	0dce
	0000	784d	2b66

Table 1: Our demo matrices P and Q for m = 4 and k = 3.

In order to reconstruct lost records in a record group, we gather the columns of **P** corresponding to available records in a matrix **H**, invert **H** with the Gaussian algorithm, multiply each available record with a coefficient of **H** and XOR the results together to obtain a missing record. Since the inversion is done once for all records to be reconstructed, it does not constitute a significant overhead.

The file starts with one data bucket and $K \geq 1$ parity buckets. The K value, called Intended Availability Level, is a file parameter. It scales up through data buckets splits, as the data buckets get overloaded. Each bucket group has then the availability level k that is K or $K - 1$, [LMS04]. Each data bucket contains a maximum number of b records. The value of b is the bucket capacity. When the number of records within a data bucket exceeds b, the bucket alerts the coordinator, a special entity coordinating splits. The latter designates a data bucket to split.

3. System Architecture

The goal of the prototype is to tune and experimentally determine LH^*_{RS} performance. Our current LH^*_{RS} implementation is described in depth in [M04]. It completes and improves that in [L00][ML02].

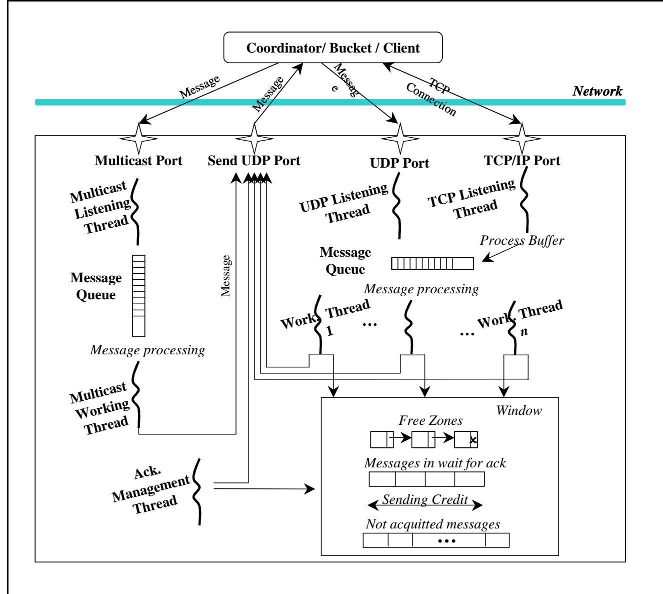


Figure 2: Bucket Architecture.

Figure 2 shows the multithreaded architecture of a bucket. Initially, a bucket is either connected to the *data buckets multicast group* or to the *parity buckets multicast group*. It starts with the multicast listening thread and the multicast working thread. When it receives a multicast group inviting the bucket to be a new or a spare bucket, it instantiates the other threads, responds positively to the coordinator, and waits for the confirmation. A selected bucket, upon receiving the confirmation, disconnects from its multicast group. Non-selected buckets cancel the instantiation process, and can commit to other invitations. Here are the functions of each thread.

- The *Multicast Listening Thread* is a temporary thread that listens to a fixed data or parity multicast port, and queues multicast messages.
- The *Multicast Working Thread* is also a temporary thread that processes queued multicast messages.
- The *UDP Listening Thread* listens to a fixed UDP port, calculated from the bucket number.
- The *Working Threads*, usually four, process queued UDP messages.
- The *TCP Listening Thread* accepts and handles multiple TCP/IP connections.
- The *Acknowledgement Manager Thread*: Each UDP message to be acknowledged is

added to the *messages in wait for the ack. list,* from which it is removed when the acknowledgement is received. This thread scans the list periodically, checks the send time of each message, and resends the message if necessary, provided that the maximum number of resends is not exceeded. In the latter case, it removes the message. Two cases may then happen. Either the sender commits an addressing error or the receiver failed. In both cases, the thread informs the coordinator.

4. Demonstration Outline

We coded our prototype in C. It includes a data and parity storage manager, and a query manager. The demonstration shows the use of LH^*_{RS} as a highly available distributed data storage system. The focus is to show how an LH^*_{RS} file scales up and how it recovers from a multiple bucket unavailability. We show the following operations.

(a) Creation of a *K*-available LH^*_{RS} file. We show how a data bucket is split, and how updates propagate to parity buckets. New data buckets are chosen from data buckets connected to the *data buckets multicast group*.

(b) Increase of the high availability of a group, by adding parity buckets. We show the interactions between the new parity bucket and its data buckets group. Each newly created parity bucket is chosen among the parity buckets connected to *parity buckets multicast group*.

(c) Recovery of *k* buckets in the group, into which we introduce failures.

(d) Key search directed to an unavailable bucket. We show that when search time-out elapses, the client alarms the coordinator. The latter checks if it is an addressing error or if the bucket is unavailable. In the latter case, the coordinator starts the recovery process.

(e) Bucket recovery operation . First, the coordinator designates a parity bucket as a recovery manager. The latter recovers records by *slices* of a given size *s*. It requests *s* successive records from each of the *m* data/parity buckets, and recovers the *s* record groups. Then, it requests the next *s* records from each bucket. While waiting, it sends the recovered slice to the spare(s). We show interaction between the coordinator, the spare data buckets, the recovery manager and the available buckets in the group.

(f) Finally, we issue search queries or display the contents of the recovered buckets, to show the recovery operation.

We also show other functions of the prototype: key search queries in normal mode, update queries, their propagation to parity buckets, record recovery using UDP, bucket recovery through UDP, we display data and parity bucket content, various statistics, etc.

Along the demonstration, we show the actual performance factors. These are basically various execution times proving the rapidity of various manipulations. We now resume those we have measured as the basis, on the original configuration [M03] [LMS04] [M04].

5. Performance Results

The hardware test bed consisted of six machines; each one has 512 MB of RAM, with a 1.8GHz Pentium processor under Windows 2K. All the machines were connected to a regular Ethernet configuration with a max bandwidth of 1 Gbps.

For the experimental set up, the record size was (and is) set to 100 bytes and the group size is set to 4 buckets. Performance results degrade for higher values of record size and group size. The best obtained performance results use Reed Solomon codes over the Galois Field $GF(2^{16})$. We use this field in our demonstration.

The time to create an LH^*_{RS} file of 25000 records was 7.896 sec for k = 0, 9.990 sec for k = 1 and 10.963 sec for k = 2. The related average times per record inserted were, 0.32 ms, 0.41 ms, and 0.44 ms for *k* = 0, 1, 2 respectively.

The average individual and bulk search times were 0.2419 ms and 0.0563 ms respectively.

Table 2 presents creation times for a parity bucket (PB) of 31 250 records.

	Total Time	Processing Time	Communication Time
1PB-XOR	2.062	1.484	0.322
1PB-RS	2.103	1.531	0.322

Table 2: Parity bucket creation times in seconds.

To measure the recovery performance, we simulated the creation of an LH^*_{RS} group with 4 data buckets and 1, 2, or 3 parity buckets. The group contained 125 000 = 4 * 31 250 data records. The recovery of a single data bucket (DB) uses the first parity bucket and consequently the XOR decoding only. The first line of Table 3 presents this case. Alternatively, the recovery can use another parity bucket, applying the RS decoding (with XORing and Galois field multiplications). The second line of the table shows the measurements for this case. Our numbers prove the efficiency of the LH^*_{RS} bucket recovery mechanism. It takes only 1.555 seconds to recover 9.375 MB of data in three buckets.

	Total Time	Processing Time	Communication Time
1DB-XOR	0.720	0.265	0.414
1DB-RS	0.855	0.380	0.400
2 DBs	1.162	0.600	0.434
3 DBs	1.555	0.911	0.464

Table 3: Data bucket recovery times in seconds.

6. Conclusion

Our demonstration shows the prototype implementation of LH*$_{RS}$: a highly available distributed data structure. We show how it actually functions. The efficient distributed storage system that our prototype constitutes can benefit modern data intensive applications: databases, grids, P2P files… Further work, in progress, concerns various aspects of current implementation, evaluation of other encoding and decoding techniques, and the applications of the prototype.

References

[B00] F. Bennour, *Un Gestionnaire de Structures Distribuées et Scalables pour les multiordinateurs Windows: Fragmentation par Hachage*, PhD thesis in French, Paris Dauphine University, 2000.

[BB94] M. Blaum, J. Brady, J. Bruck & J. Menon, *EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures*, IEEE 1994.

[BK95] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby & D.Zuckerman, *An XOR-Based Erasure-Resilient Coding Scheme*, ICSI Tech. Rep. TR-95-048, 1995.

[H94] L. Hellerstein, G.A. Gibson, R.M. Karp, R.H. Katz & D.A. Patterson, *Coding Techniques for handling Failures in Large Disk Arrays*, Algorithmica, 1994, 12, pp.182-208.

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

[LS00] W. Litwin & J.E. Schwarz, LH*$_{RS}$, *A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes*, p.237-248, Proceedings of the ACM SIGMOD 2000.

[LMS04] W. Litwin, R. Moussa & J.E. Schwarz, *LH*$_{RS}$ – A Highly-Available Scalable Distributed Data Structure*, CERIA Res. Rep. May 2004.

[ML02] R. Moussa & W. Litwin, *Experimental Performance Management of LH*$_{RS}$ *Parity Management*, Distributed Data and Structures, 4, (WDAS02) Carleton Scientific, Waterloo, Ontario, CA. 2003, pp. 87-98.

[M03] R. Moussa, *Experimental Performance Analysis of the new LH*$_{RS}$ *Scenarios and Architecture Design*, CERIA Res. Rep., June 2003,http://ceria.dauphine.fr/Rim/comparison0603.pdf.

[M04] R. Moussa, *Contribution à l'étude des Structures de Données Distribuées et Scalables à Haute disponibilité*, PhD thesis in French, Paris Dauphine University, 2004.

[MS04] R. Moussa & J.E. Schwarz, *Design and Implementation of LH*$_{RS}$*: a Highly Available Distributed Data Storage System*, Workshop on Distributed Data and Structures (WDAS04).

[P97] J. S. Plank, *A Tutorial on Reed-Solomon Coding for fault-Tolerance in RAID-like Systems*, Software – Practise & Experience, 27(9), Sept. 1997, pp 995- 1012.

[PGK88] D. A. Patterson, G. Gibson & R. H. Katz, *A Case for Redundant Arrays of Inexpensive Disks*, Proc. of ACM SIGMOD Conf, pp.109-106, June 1988.

[R89] M. O. Rabin, *Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance*, Journal of ACM, Vol. 26, N° 2, April 1989, pp. 335-348.

[S02] T. J.E. Schwarz S.J., *Reed Solomon Codes for Erasure Correction in SDDS*, Distributed Data and Structures, 4, (WDAS02) Carleton Scientific, Waterloo, Ontario, CA. 2003.

[SDDS] http://ceria.dauphine.fr/SDDS-bibliographie.html

[XB99] L. Xu & J. Bruck, *Highly Available Distributed Storage Systems*, Proceedings of workshop on Distributed High Performance Computing, Lecture notes in Control and Information Sciences, Springer Verlag, 1999.

[WK02] H. Weatherspoon & J. D. Kubiatowicz, *Erasure Coding vs. Replication: A quantitative Comparison*, Proceedings of the 1st International Workshop on Peer-to-Peer Systems, March 2002, p.328-338.

Acnowledgements

This work was partly supported by the European Commission project ICONS (*project no. IST-2001-32429*), Microsoft Research, as well as by a scholarship from the Tunisian government.

Design and Implementation of LH*$_{RS}$: a Highly Available Distributed Data Storage System

Rim Moussa
CERIA Lab., Université Paris Dauphine
FRANCE
Rim.Moussa@dauphine.fr

Thomas J.E. Schwarz, S.J.
Santa Clara University
USA
TSchwarz@calprov.org

Abstract

The ideal storage system is always available and is incrementally expandable. Existing storage systems are far from this ideal. Affordable computers and high-speed networks allow us to investigate storage architectures that bring us closer to the ideal storage system. We describe a prototype implementation of the highly available scalable distributed data structure LH*$_{RS}$. The scheme allows to recover from a multiple unavailability using a variant of Reed Solomon erasure correcting code. We present the system architecture and experimental performance measurements.

Keywords: High availability, Scalable and Distributed Data Structures, Reed Solomon Codes, Erasure-resilient systems.

1 Introduction

The Scalable and Distributed Data Structures [SDDS] are being developed for computers over fast networks, usually local networks, i.e. for the *multicomputers.* This new hardware architecture is promising and becomes highly popular. In spite of the advantages given by the data distribution layout, vulnerability to failures remains the arena, and accentuates with the increase of the number of machines in the network. Many approaches to build highly available, i.e., fault tolerating, distributed data storage systems have been proposed. They generally fall into the two categories of (*i*) data mirroring, and (*ii*) parity information. In the former approach the storage overhead is prohibitive. The latter approach uses erasure-correcting codes to guard against failures. The simplest codes, e.g. in RAID systems [PGK88], use XOR calculus for the tolerance of a single site failure. For multiple failures more complex codes are needed. These can be the binary codes [H94] for double or triple failure, or character codes. Examples of character codes are: the array codes as the EVENODD code [BB94], the X-code [XB99] or the Reed Solomon codes. The latter appear best at present to deal with multiple failures [R89, BK95, P97, LS00, M00, ML02]. Theoretical proofs demonstrating superiority of erasure resilient systems to replicated systems can be found in [S02, WK02].

Below, Section 2 recalls the LH*$_{RS}$ file structure. Section 3 overviews a proposed architecture for LH*$_{RS}$. Performance results are given in section 4. Finally, section 5 concludes the article.

2 LH*$_{RS}$ Scheme

LH*$_{RS}$ scheme is described with details in [LS00, S02, ML02]. An LH*$_{RS}$ file is subdivided into groups. Each group is composed of *m* Data Buckets and *k* Parity Buckets. The data buckets store the data records, same for parity buckets. Every data record fills a rank *r* in its data bucket. A record group consists of all records with the same rank in a bucket group. We construct parity records from data records having the same rank within data buckets forming a bucket group (see Fig. 1(a)). The record grouping has an impact on the data structure of a parity record. The latter keeps track of the data records it is computed from. Fig. 1(b-c) shows each of the structure of a data record and a parity record. The parity calculus is done using Reed Solomon codes.

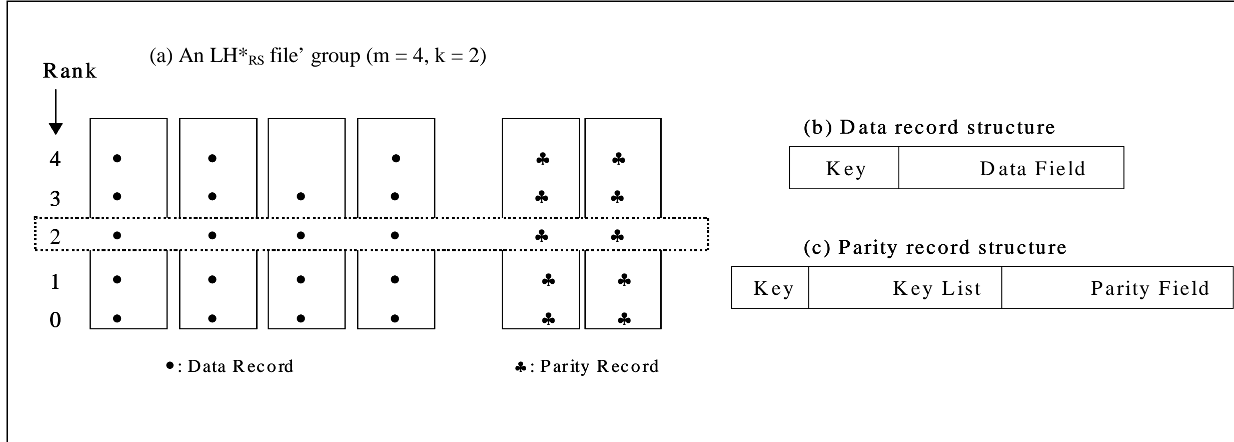


Figure 1: LH*$_{RS}$ file Structure

In the scenarios described below, all the buffers are sent through TCP/IP for performance and reliability concerns demonstrated in [M00, ML02], where we compared TCP/IP-based scenarios to UDP-based scenarios.

2.1 Data Bucket split Scenario

The file starts with one data bucket and *k* parity bucket. It scales up through data buckets' splits, as the data buckets get overloaded. Each data bucket contains a maximum number of *b* records. The value of *b* is the bucket capacity. When the number of records within a data bucket exceeds *b*, the bucket adverts a special entity coordinating the splits, which is the coordinator. The latter designates a data bucket to split.
During a data bucket split process, half of the splitting data bucket contents move to a new created data bucket. As a consequence to the data records' transfer, the data records remaining in the splitting data bucket and those moving get new ranks. So, the parity buckets belonging to (*i*) *g1:* the splitting data bucket's group, and (*ii*) *g2:* the new data bucket's group, have to be updated. In that way during the split process, two update buffers are filled respectfully at the splitting data bucket and the new data bucket, and sent to update the parity buckets of each group.

2.2 High availability Scenario

The high availability scenario ensures the increase of the availability of a group, just by adding a parity bucket. The new parity bucket executes at most $x \leq m$ stages, such that x is the number of not dummy data buckets in the group *g*. At each stage, the parity bucket updates its contents with respect to each data bucket contents. Each data bucket of the group adds the new parity bucket to the list of its group reliability, and will be able to reflect the client's manipulations on this parity bucket.

2.3 Bucket recovery Scenario

The data buckets' recovery scenario starts at the coordinator level. The coordinator probes all buckets belonging to the group, waits a time-out, then either notifies the application of the impossibility of doing recovery, due to the lack of surviving buckets; or assigns to the first parity bucket replying to the probe the task of failed buckets' recovery. In case of possible recovery, the elected parity bucket is adverted of the bucket's states and addresses. It chooses *m* buckets among surviving one's, in preference parity buckets. That is to let the data buckets attend to application requests.
At each iteration, the recovery manager asks participating buckets to search *slice* records, corresponding to ranks in range [*r*, …, *r*+*slice*-1]. Then, *r* is incremented of slice. The buckets reply in the limit of the number of records they hold. And, on the receipt of the buffers, data records having same rank are retrieved from the buffers and from the local data structure, to compute missing records. Finally, the recovery manager sends to each spare data bucket its partial recovered contents.

3 System Architecture

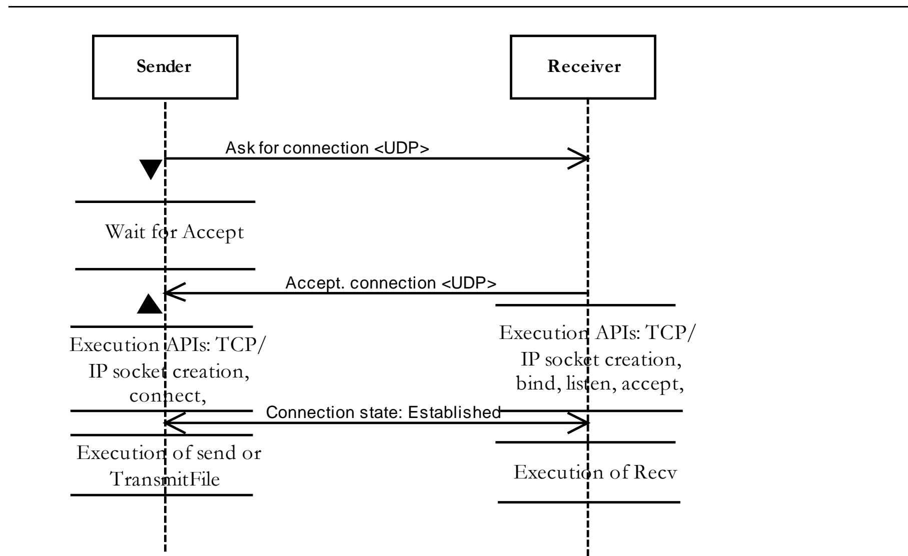
In [M00][ML02], we have implemented our scenarios related to file creation –data bucket split, high availability and buckets recovery, on the top of SDDS2000 architecture [D01]. SDDS2000 architecture was proposed by F. Bennour and A. W. Diène, as an architecture for LH* and RP*. In order to have better performance results, we embedded to SDDS2000 other components, namely: (*i*) an efficient TCP/IP connections handler, (*ii*) flow control and acknowledgements strategy, and (*iii*) a dynamic addressing structure.

To M. Welch and al. [WGBC00] classification of server architectures, our architecture is an hybrid architecture. Indeed, it falls in the spectrum between multithreaded architectures and event-driven architectures, since it combines multithreaded architectures and queues. The threads communicate through events and queues, and concurrent threads are synchronized using common concurrency-programming tools.

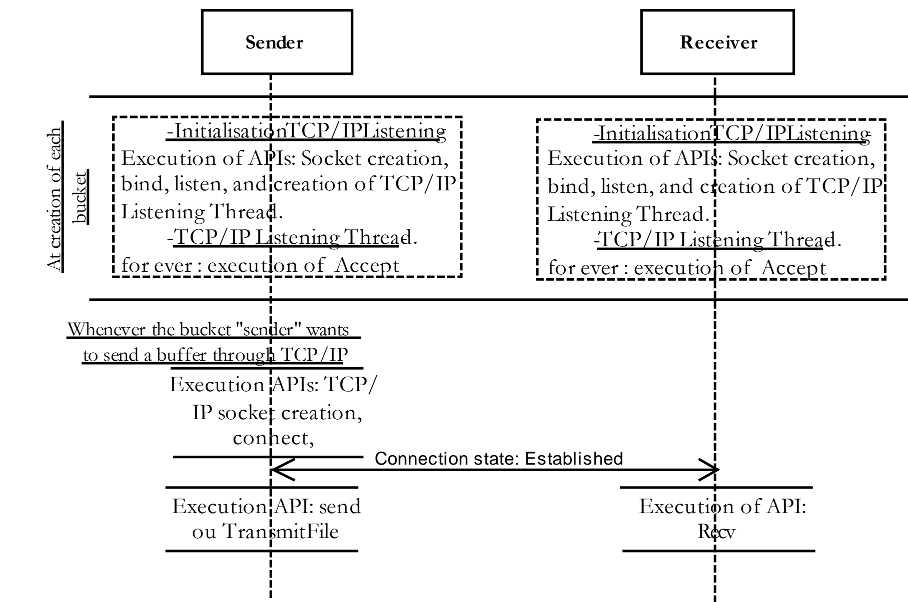
3.1 TCP/IP connections handler

In our last implementation of LH^*_{RS} scenarios mapped to SDDS2000 architecture [ML02], the communication time dominates the total time; especially for TCP-based scenarios, i.e., the parity bucket creation scenario and the buckets recovery scenario. To improve the performance results, we have enriched SDDS2000 architecture with an efficient TCP/IP connections handler, and mapped our scenarios to the new architecture.

According to RFC 793 [ISI81] and [MB00], we can open TCP/IP connections in a passive OPEN way, i.e, a process will accept and queue incoming connection requests. The *backlog* parameter designates the number of pending TCP/IP connections. The Windows Sockets 1.1 specification indicates that the maximum allowable value for a backlog is 5; however, Windows 2000 Server accepts a backlog of 200, and Windows 2000 Professional accepts a backlog of 5.
Figure 2, details the way we establish a TCP/IP connection in both of SDDS2000 architecture and the new devised architecture. In SDDS2000, in order to establish a TCP/IP connection, first two messages sent through UDP are exchanged between the two peers. Added to that overhead, the delay underwent to establish the connection while each peer executes appropriate APIs. In our architecture, a TCP listening thread is instantiated on the bucket creation, and handles any incoming connection. Likewise, we don't need to synchronize the peers to establish a TCP/IP connection between them, since in both sides TCP/IP connections are passive OPEN, and the 'sender' peer executes appropriate APIs, without asking the 'receiver' peer to get ready.



(*a*) TCP/IP Connection in SDDS2000



(*b*) TCP/IP Connections Handler

Figure 2: TCP/IP connection handler.

3.2 Flow control and acknowledgement strategy

Diéne [D02] proposed a flow control and acknowledgement strategy, to prevent messages losses under UDP protocol. With respect to his strategy, we designed our flow control and acknowledgement strategy that deploys only one additional thread, while Diéne's strategy deploys *window size+2* threads. The *Window size* parameter is the number of messages that could be sent without acknowledgements, such that each thread handles one message at a time.

For that purpose, each peer has a *Sending Credit* (or *Window size*), that when it reaches zero, the sender stops sending messages, else the sending process pulls a free position from a FIFO managed *Free Positions Queue*, adds the message to *Not Yet Acquitted Messages List*. The message is obviously

removed when the corresponding acknowledgement is received; consequently a new position is signaled free, and queued to *Free Positions Queue*. The *Acknowledgement manager Thread* scans periodically the list, checks sending time of each message, and re-sends if necessary the message whenever the maximum number of re-sends is not exceeded. In the last case, the message is removed, and two cases are considered. Indeed, either the '*sender'* peer commits an addressing error or the '*receiver'* peer is failed. In all cases, the coordinator is informed.

3.3 Dynamic addressing structure

In our first implementation, a static table containing the IP addresses of the different involved peers: clients, data/ parity buckets, is used for addressing purpose. We proposed a new scenario to add a data/parity bucket to the file on-line, so that the addressing table evolves accordingly to the file and is not user-fixed.
For that purpose, a bucket depending on its type data or parity bucket, is either connected to the *data buckets multicast group* or the *parity buckets multicast group*. It starts with the *multicast listening thread* and the *multicast working thread*. The former listens to a fixed data or parity multicast port, and queues multicast messages, and the latter processes queued multicast messages. When the bucket receives a multicast message inviting the bucket to be a new or a spare bucket, it instantiates the other threads, responds positively to the coordinator, and waits for the confirmation. A selected bucket, upon confirmation receipt, disconnects from its multicast group, while the non-selected buckets cancel the instantiation process, and can commit to other invitations.
The new bucket selection scenario, has without doubt changed our architecture, and is applied to each of data bucket split scenario, high availability scenario and bucket recovery.

3.4 Architecture

Hereafter, we briefly describe each functional thread, the overall bucket architecture is illustrated in Figure 3.

Multicast Listening Thread: is a temporary thread. The thread listens to a fixed data or parity multicast port, and queues multicast messages.

Multicast Working Thread: is also a temporary thread. It processes queued multicast messages.

UDP Listening Thread: listens to a fixed UDP port. The latter is deduced from the bucket number.

Working Threads: a working thread processes queued UDP messages.

TCP Listening Thread: accepts and handles multiple TCP/IP connections.

Acknowledgement manager Thread: scans the *Not Yet Acquitted Messages List*, checks *sending time* of each message, and re-sends if necessary the message whenever the maximum number of re-sends is not exceeded, or deletes the message other way.

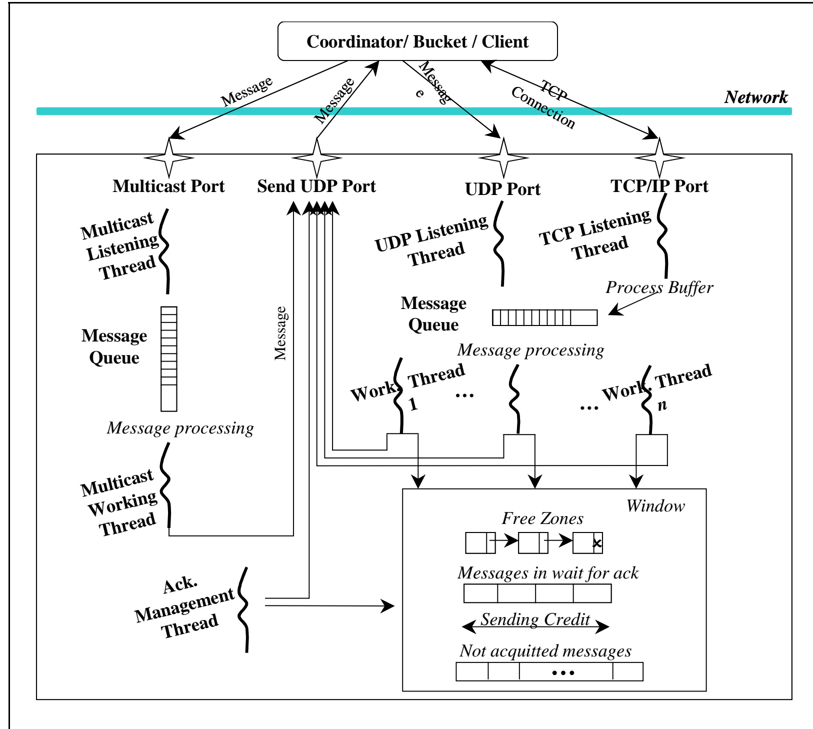


Figure 3: Bucket Architecture

4 Performance Results

The goal of the prototype is to tune and experimentally determine the LH^*_{RS} performance characteristics. Our LH^*_{RS} implementation improves that presented in [ML02]. It uses distributed RAM memory, and includes a data and parity storage manager, and a basic query manager. Other functionalities are implemented as key-based search queries, data update queries and propagation of updates to parity buckets, record's recovery using UDP, buckets' recovery through UDP, display bucket contents and statistics, etc.

The hardware testbed consists of six machines; each one has 512 MB of RAM, with a 1.8GHz Pentium processor, and runs Windows 2K Server. All the machines are connected to a regular Ethernet configuration with a max bandwidth of 1Gbps. For the experimental set up, the *Record Size* is set to 100 bytes and the *Group Size* is set to 4 buckets. Performance results are expected to degrade for higher values of *Record Size* and *Group Size*. The best obtained performance results are using Reed Solomon codes with encoding and decoding in Galois Field: $GF(2^{16})$. Experiments details and a full comparison between different architectures and configurations can be found in [M03].
Our Generator matrix is constructed such that its first column is filled with '1's, this reduces Galois field multiplication to simple XOR calculus. Consequently, (*i*) the first parity bucket of each group is XOR-encoded, and (*ii*) in case of one data bucket failure and the first parity bucket is alive, the bucket is recovered using XOR-decoding.

4.1 File creation

Along a synchronous inserts, where the client waits for a reply before issuing the next insert query, the time to create an LH^*_{RS} file of 25000 records is 7.896 sec for $k = 0$, 9.990 sec for $k = 1$ and 10.963 sec for $k = 2$. We get better performance results, with the new flow control and acknowledgement strategy

described in §3.2. Indeed, for *Window Size* fixed to 5, the time to create an LH*$_{RS}$ file of 25000 records is 4.484 sec for $k = 0$, 6.969 sec for $k = 1$ and 8.109 sec for $k = 2$.

4.2 Search Performances

The key search time is the basic referential of access performance of the prototype, since it does not involve the parity calculus for $k > 0$. We have measured the time to perform random *individual* (synchronous) and *bulk* (asynchronous) successful key searches. For synchronized searches, the client waits for a reply before issuing another search query. While in asynchronous searches, the client sends a flow of search queries to four data buckets. All measures were at the client side.
We have measured the search times in a file of 125000 records, distributed over four buckets. The average individual and bulk search times were 0.2419 ms and 0.0563 ms respectively. Thus the former is about 40 times faster than a disk key search. The latter reaches the speed-up of almost 200 times. The former was bound basically by the server processing speed, while the latter by the client speed.

4.3 Data Record Recovery

The record recovery manager is located at one of the parity buckets. First, it looks for the data record key inside the parity bucket structure, sends search queries to alive buckets, then waits until receipt of replies to compute the missing record, finally it sends the recovered record to the client.
We have measured the recovery times in a file of 125000 records, distributed over four buckets. The timing is measured at the parity bucket and starts when the bucket gets the message from the coordinator, until the recovery of the record. The average data record recovery time is 1.30 ms using XOR decoding and 1.32 using RS decoding. Notice that, the average time to scan our parity bucket to locate the key c of the data record was measured to be 0.822 ms. This is the dominant part of the total time as it represents 62% and 64% respectively. If one seeks for faster record recovery, or buckets are much larger, the additional already mentioned index (c, r) per data bucket at the parity bucket should help. Notice finally that even the basic record recovery times remain significantly faster than for a disk file.

4.4 High Availability

To measure the recovery performance, we create an LH*$_{RS}$ group with 4 data buckets, the group contained 125 000 = 4 * 31 250 data records. Table 1 presents parity bucket (PB) creation times.

	Total Time	*Processing Time*	*Communication Time*
1PB-XOR	2.062	1.484	0.322
1PB-RS	2.103	1.531	0.322

Table 1: Parity bucket creation times in seconds.

Notice that, (*i*) the time to create the first parity bucket (PB-XOR), using XORing only, is faster than for the other buckets (PB-RS), using the RS calculus, and (*ii*) the communication time represents almost 15% of the total time, while in [M00][M02] it represents 60% of the total time.

4.5 Buckets Recovery

To measure the recovery performance, we create an LH*$_{RS}$ group with 4 data buckets and 1, 2, or 3 parity buckets. The group contained 125 000 = 4 * 31 250 data records. The *Slice* parameter varies in the set {1250, 3125, 6250, 15625, 31250}, being respectively {4%, 10%, 20%, 50%, 100%} of a bucket's size.
The recovery of a single data bucket (DB), can use the first parity bucket and consequently the XOR decoding only. The 1st line of Table 2 presents this case. Alternatively, the recovery can use another parity bucket, applying the RS decoding). The 2nd line of the table shows the measures of this case. Our numbers prove the efficiency of the LH*$_{RS}$ bucket recovery mechanism. It takes only 1.555 seconds to recover 9.375 MB of data in three buckets.

	Total Time	*Processing Time*	*Communication Time*
1DB-XOR	0.720	0.265	0.414
1DB-RS	0.855	0.380	0.400
2 DBs	1.162	0.600	0.434
3 DBs	1.555	0.911	0.464

Table 2: Data bucket recovery times in seconds.

During experiments, in the set of *Slice* parameter values {1250, 3125, 6250, 15625, 31250}, the recovery performances are almost equal. Table 2 reports the average times, for experiments details refer to [M03].

5 Conclusion

We have evaluated an $LH*_{RS}$ file creation time, parity bucket creation, data retrieval in both normal mode and degraded mode, and finally the recovery of more than one data bucket, with respect to the new bucket architecture. Thanks to the TCP/IP connections handler component, communication times are improved of 80% [M03], and no more dominates total times for TCP/IP based scenarios. Experiments prove the efficiency of the proposed scenarios to $LH*_{RS}$ scheme and validate our devised architecture.

Our architectural proposals and scenarios are independent of the erasure resilient code used and data distribution scheme. Further work concerns implementation of other encoding and decoding techniques.

References

[B00] F. Bennour, *Un Gestionnaire de Structures Distribuées et Scalables pour les multiordinateurs Windows: Fragmentation par Hachage*, PhD thesis in French, Paris Dauphine University, 2000.

[BB94] M. Blaum, J. Brady, J. Bruck & J. Menon, *EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures*, IEEE 1994.

[BK95] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby & D.Zuckerman, *An XOR-Based Erasure-Resilient Coding Scheme*, ICSI Tech. Rep. TR-95-048, 1995.

[D01] A W. Diène, *Contribution à la Gestion de Structures Distribuées et Scalables*, PhD thesis in French, Paris Dauphine University, 2002.

[H94] L. Hellerstein, G.A. Gibson, R.M. Karp, R.H. Katz & D.A. Patterson, *Coding Techniques for handling Failures in Large Disk Arrays*, Algorithmica, 1994, 12, pp.182-208.

[ISI81] Information Sciences Institute, RFC 793: *Transmission Control Protocol (TCP) – Specification*, Sept. 1981, http://www.faqs.org/rfcs/rfc793.html.

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

[LS00] W. Litwin & J.E. Schwarz, $LH*_{RS}$, *A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes*, p.237-248, Proceedings of the ACM SIGMOD 2000.

[M00] R. Moussa, *Implantation partielle & Mesures de performances de $LH*_{RS}$*, Université Paris Dauphine, MSc Report in French, October 2000, http://ceria.dauphine.fr/Rim/dea.pdf.

[M03] R. Moussa, *Experimental Performance Analysis of the new $LH*_{RS}$ Scenarios and Architecture Design*, CERIA Research Report, June 2003, http://ceria.dauphine.fr/Rim/comparison0603.pdf

[MB00] D. MacDonal, W. Barkley, *MS Windows 2000 TCP/IP Implementation Details*, http://secinf.net/info/nt/2000ip/tcpipimp.html.

[ML02] R. Moussa & W. Litwin, *Experimental Performance Management of $LH*_{RS}$ Parity Management*, WDAS 2002 proceedings, pp. 87-98.

[P97] J. S. Plank, *A Tutorial on Reed-Solomon Coding for fault-Tolerance in RAID-like Systems*, Software – Practise & Experience, 27(9), Sept. 1997, pp 995- 1012.

[PGK88] D. A. Patterson, G. Gibson & R. H. Katz, *A Case for Redundant Arrays of Inexpensive Disks*, Proc. of ACM SIGMOD Conf, pp.109-106, June 1988.
[R89] M. O. Rabin, *Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance*, Journal of ACM, Vol. 26, N° 2, April 1989, pp. 335-348.
[S02] T. J.E. Schwarz S.J., *Reed Solomon Codes for Erasure Correction in SDDS*, WDAS 2002 proceedings, pp. 75-86.
[SDDS] http://ceria.dauphine.fr/SDDS-bibliographie.html
[WGBC00] M. Welch, S. D. Gribble, E. A. Brewer, D. Culler, *A Design Framework for Highly Concurrent Systems*, UC Berkeley Tech. Report UCB/CSD-00-1108, April 2000.
[WK02] H. Weatherspoon & J. D. Kubiatowicz, *Erasure Coding vs. Replication: A quantitative Comparison*, Proceedings of the 1st International Workshop on Peer-to-Peer Systems, March 2002, p.328-338.
[XB99] L. Xu & J. Bruck, *Highly Available Distributed Storage Systems*, Proceedings of workshop on Distributed High Performance Computing, Lecture notes in Control and Information Sciences, Springer Verlag, 1999.

Experimental Performance Analysis of LH*$_{RS}$ Parity Management

RIM MOUSSA
CERIA Lab.-Université Paris dauphine, France

WITOLD LITWIN
CERIA Lab.-Université Paris dauphine, France

Abstract

We present an implementation of LH*$_{RS}$: a high-availability Scalable Distributed Data Structure on Windows 2000. We show experimental determination of the time for £le creation, key search, record recovery, parity bucket creation, and data bucket recovery. The measures prove the ef£ciency of the LH*$_{RS}$ scheme, the scalability especially.

Keywords

High Availability, Fault-tolerance, Scalability, Distributed Data Structures, Reed Solomon codes, Multicomputers

1 Introduction

Scalable Distributed Data Structures [SDDS] are being developed for computers over fast networks, usually local networks, i.e. for the multicomputers. This new hardware architecture is promising and is becoming highly popular. In spite of the advantages given by the data distribution layout, failures remain the problem, and the vulnerability accentuates with the increase of the number of machines in the network. Many approaches to build highly available, i.e., fault tolerating, distributed data storage systems have been proposed. They generally fall into the two categories of (i) data mirroring and (ii) parity information. The latter approach uses erasure correcting codes to guard against failures The simplest codes, e.g. in RAID systems, use XOR calculus for the tolerance of a single site failure. For multiple failures more complex codes are needed. These can be the binary codes [H94] for double or triple failure, or character codes. Examples of character codes are: the array codes as the EVENODD code [BB94], the X-code [XB99] or the Reed Solomon codes. The latter appear best at present to deal with multiple failures [R89] [BK95] [P97] [LS00].

The Scalable and Distributed Data Structure (SDDS) schema called [LS00] provides high-availability server nodes of data buckets by using Reed Solomon codes. We present our contribution to the design and implementation of a prototype system for LH*$_{RS}$ £les on Windows 2000 multicomputers. The goal of the prototype is to tune the system and experimentally determine the LH*$_{RS}$ performance. Our LH*$_{RS}$ implementation completes and improves the one presented in [L00]. The entire prototype reuses an existing LH* implementation that does not provide the high-availability [B00]. Our contribution to the prototype design consists in the solutions to the problems encountered by the £rst prototype, namely, (1) a more reliable data bucket split, (2) a more ef£cient data buckets'recovery, and (3) a parity bucket creation algorithm increasing the availability of a data bucket group. We show experimental determination of the time for £le creation, key search, record recovery, parity bucket creation, and data bucket recovery.

Below, Section 2 recalls the LH*$_{RS}$ £le structure. Section 3 overviews our contribution to the implementation. Further sections present the performance analysis. Section 8 concludes the article.

The hardware testbed consists of six machines; each has 128 MB of RAM, with a 731MHz Pentium III processor, and runs Windows 2000. All the machines are connected by a 100 Mbps Ethernet. For the experiments, the record size is set to 100 bytes and the con£guration tested consists of one client, four data buckets and k parity buckets; $k \in \{0, 1, 2\}$.

2 LH*$_{RS}$ Data Structure

LH*$_{RS}$ scheme is described in detail in [LS00]. An LH*$_{RS}$ £le is subdivided into groups. Each group is composed of m data buckets and k parity buckets. The *data buckets* store the data records, while the parity buckets store the parity ones. Every data record £lls a rank r in its data bucket. A record group consists of all records with the same rank in a bucket group. We construct parity records from data records having the same rank within data buckets forming a bucket group Fig. 1a. The record grouping has an impact on the data structure of a parity record. The latter keeps track of the data records it is computed from. Fig. 1b-c, shows each the structure of a data record and a parity record. The parity calculus is done using Reed Solomon codes.

The £le starts with one data bucket and one parity bucket. It scales up through data buckets splits, as the data buckets get overloaded. Each data bucket contains a maximum number of b records. The value of b is the bucket capacity. When the number of records within a data bucket exceeds b, the bucket adverts a special entity called the split coordinator. The latter designates a data bucket to split.

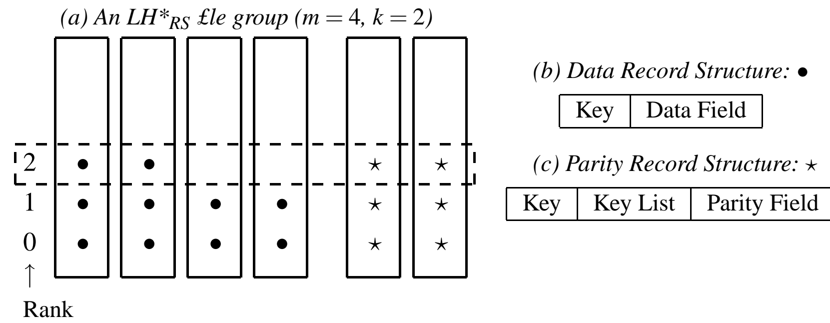


Figure 1: An LH*$_{RS}$ £le Structure.

3 File Creation

3.1 Bucket Splitting

During a data bucket split, half of the splitting data bucket move to a newly created data bucket. As a consequence to the data transfer, the records remaining in the data bucket and those moving get new ranks. The parity buckets belonging to $g1$, the splitting data bucket group, and to $g2$, the new data bucket's group, have to be updated. A £rst update scenario is proposed in [L00]. It consists in sending update messages to parity buckets. Each parity bucket belonging to $g1$ receives one message per record to delete, i.e., b delete messages in total and one message per record to insert, i.e., about $b/2$ insert messages, a total of $1.5 * b$ update messages. Same for each parity bucket belonging to $g2$, it receives $b/2$ insert messages. Those messages are sent through an unreliable protocol that is UDP without acknowledgements. The parity buckets may be inconsistent if the messages are lost, besides the communication is inef£cient since many small messages are sent.

We have implemented a splitting scenario overcoming disadvantages of the above one. We have replaced UDP by TCP/IP, sending a single message with all parity records to update. The update buffer is a collection of the structure S:

- Record's key;
- Record's Data £eld;
- Old Rank: the rank occupied by the record; and it has to be deleted from it;
- New rank: the rank of the record after the split, that it has to be inserted on it.

We now give the details of this scenario at the level of the involved buckets:

→*The splitting data bucket:*
The splitting data bucket £lls in each structure S as follows: If the data record is moving: then it is to be deleted from parity records of rank *oldrank* in the parity buckets, belonging to the same group as

the splitting data bucket. And, *newrank* is set to -1. Else, if the data record is staying: the data record is to be £rst deleted from parity records of rank *oldrank*, and then reinserted at rank *newrank*.

→*New data Bucket:*
Following a data bucket split, the new data bucket receives a buffer, containing about $b/2$ records to insert. While inserting, it prepares a buffer to update the content of its parity buckets. The parity bucket uses the structure S, the £eld *oldrank* is set to –1, and the £eld *newrank* indicates the rank occupied by the record just inserted.

→*Parity bucket:*
Each parity bucket, receiving an update buffer, processes each structure S as follows: (1)If $S.oldrank \neq 1$, delete the data record $\{S.key, S.datafield\}$ from parity record of rank *oldrank*. (2) Else if $S.newrank \neq 1$, insert the data record $\{S.key, S.datafield\}$ in the parity record of rank *newrank*.

3.2 Experimentation description

First we create a £le of 25000 records. The £le spread on four data buckets, each containing 6250 records. During the £le creation, three splits occurred. The £rst split is of the data bucket numbered 0. Then, data buckets 0 and 1 split at approximately the same time. Hereafter, Figure 2 reports on £le creation performance.

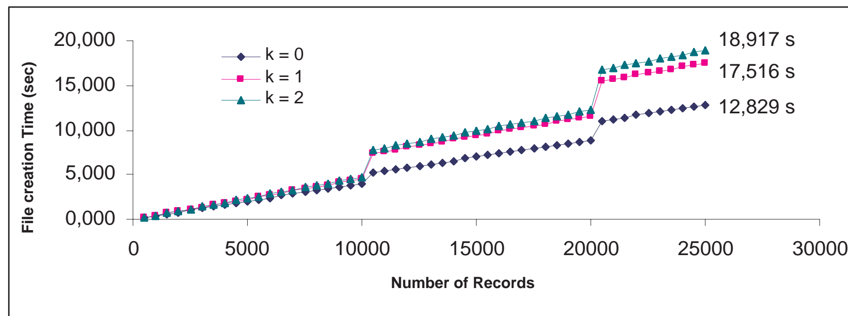


Figure 2: Performance Measurements of the creation of a k-available LH*$_{RS}$ File, that spread on 4 data buckets (25000 data records) k $\in \{0, 1, 2\}$

Following a data bucket split, there is an increase in the insert average time of 40% from $k = 0$ to $k = 1$, and of 7% from $k = 1$ to $k = 2$. The increase is mainly due to the preparation and sending of TCP/IP update buffers (case of $k \geqslant 1$) from the splitting data bucket and new data bucket to their group reliability. We

estimate that the splitting process may become cumbersome when k increases, and a more ef£cient parity buckets' update strategy is then required. One can release the splitting data bucket and the new data bucket from the task of communicating the updates to all parity buckets of its group.

To prove the scalability of the £le creation scenario, we conduct experiments £rst on the creation of an LH*$_{RS}$ £le, that spread on 8 data buckets, and containing 50000 records (see Figure 3), and another £le containing 25000 records, that spread on 4 data buckets, but has group size 2. We obtain an insert time per record approximately equal to {0.40ms, 0.44ms, 0.48ms} respectively to $k \in \{0, 1, 2\}$. We notice that the insert time per record is invariant of group size and of the number of records inserted.

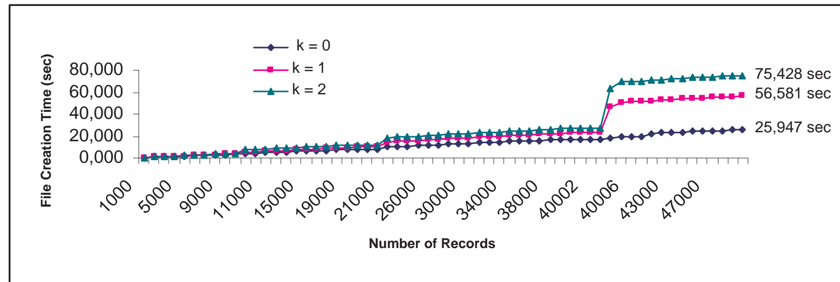


Figure 3: Performance Measurements of the creation of a k-available LH*$_{RS}$ File, that spread on 8 data buckets (50000 data records) k $\in \{0, 1, 2\}$

4 Parity Bucket Creation

The high availability scenario ensures the increase of the availability of a group, just by adding a parity bucket. We begin by describing the scenario, then we on report the time to create a parity bucket.

4.1 Creation Scenario

The new parity bucket is created in at most $x \leq m$ steps, such that x is the number of not dummy data buckets in the group g. At each step, the parity bucket establishes a TCP/IP connection with one data bucket, and the latter sends its contents. The received buffer is so processed for insert/ update.

Each data bucket of the group adds the new parity bucket to the list of its group reliability, and will be able to re¤ect the client's manipulations on this parity bucket.

4.2 Parity Bucket Creation Performances

In each experiment, we create a £le of 2.5∗Bucket Size records. The £le spreads on up to four data buckets, each of 0.625∗Bucket Size records. The performance results are in Table 1. We highlight *The Communication Time*, i.e. the time spent on establishing TCP/IP connections to receive each data bucket content; and *The Process Time*, i.e. time spent on processing the buffer.

BucketSize	*Com.Time(sec)*	*ProcessTime(sec)*	*Total(sec)*
5000	2,211	0,302	2,523
10000	2,433	0,611	3,044
25000	3,185	1,652	4,847
50000	4,667	3,395	8,062

Table 1: Parity Bucket Creation Performance Results

We notice that the time spent on establishing TCP/IP connections with the four data buckets is relatively high compared to process time. Indeed, the communication time is {87.63%, 79.93%, 65.71%, 57.89%} for buckets sizes {5000, 10000, 25000, 50000}. It appears interesting to choose a larger bucket to amortize TCP/IP connection cost.

5 Basic Key Search Performance

We measure both parallel searches, during which a client sends a ¤ow of search messages in parallel to the four data buckets, and synchronized search, where the client waits for a reply before issuing another request. Figure 4 presents the times for a £le of size 125000 distributed over 4 buckets.

The search times per record are essentially independent of the number of searches. The synchronized search time measures the server access time. Let's notice that it is 30 times faster than that to a £le on a typical local disk. The parallel search time measures the maximal client speed. Notice also that this search time is about 100 times faster than that to a local disk.

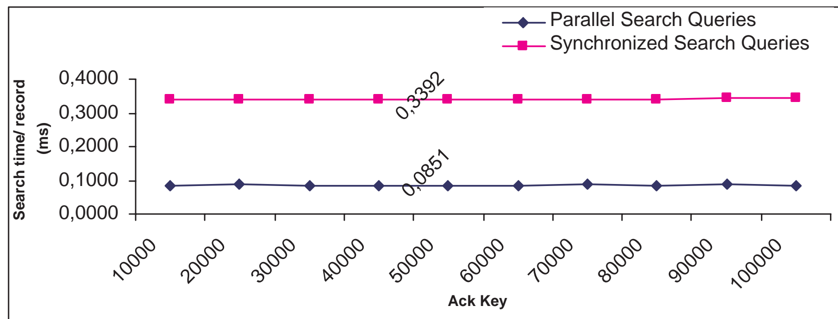


Figure 4: Search Performance Results

6 Data Record Recovery

6.1 Recovery Algorithm

The routine "recover one record" follows the following steps:

1. Look for the data record's key inside the parity bucket structure,
2. Send search queries to alive buckets,
3. Wait until receiving replies to sent queries,
4. Compute the missing data record through RS-decoding,
5. Send the recovered record to the client.

6.2 Performance Analysis

We create an LH*$_{RS}$, 1-available £le of 125000 records. Then, we simulate the failure of one data bucket. Figure 5 summarizes the performance results of recovering x records of the failed data bucket.

The time to look for a data record's key inside a parity bucket structure, of 32150 records, is approximately equal to $0.7ms$. It is 56% of the average time to recover a data record.
Notice that while the record recovery time is 15 times higher than that of a normal key search, it is still 8 times faster than a normal key search on a local disk.

7 Data Bucket Recovery

We describe and we report the performance results of two scenarios devised.

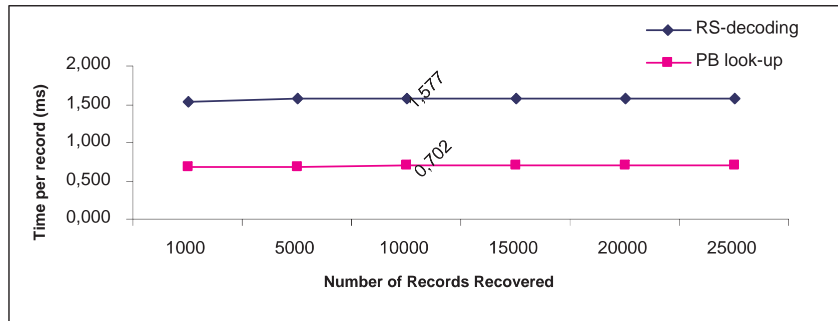


Figure 5: Data Records Recovery Performance Results

7.1 UDP-based Data Buckets' recovery Scenario

The data buckets' recovery scenario starts at the coordinator level. The coordinator probes all buckets belonging to the group, waits a time-out, then either noti£es the application of the impossibility of recovery, due to the lack of surviving buckets; or assigns to the £rst parity bucket replying to the probe the task of failed buckets' recovery. In the case of recovery, the elected parity bucket is adverted of the bucket's states and addresses. It chooses m buckets among surviving ones, parity buckets being preferred. This is to let the data buckets being available for the application requests, while the recovery is in progress.

To measure the performance of the scenario, we create a k-available LH*$_{RS}$ £les, with $k \in \{1, 2, 3\}$. The created £le spreads over four data buckets. At each experiment, we simulate the failure of k data buckets, and then we recover them. Table 2 reports decoding performance for the recovery of k data buckets, such that k $\in \{1, 2, 3\}$ for different bucket sizes.

Bucket Size	Bucket contents	1 DB recovery (s)	2 DBs recovery (s)	3 DBs recovery (s)
5000	3125	1.742	2.043	2.564
10000	6250	3.465	4.076	4.797
25000	15625	8.662	10.185	11.928
50000	31250	17.375	20.349	23.895

Table 2: Performance Measurements of k Data Buckets Recovery using UDP protocol, $k \in \{1,2,3\}$.

Figure 6 proves the scalability of the algorithm. Independently of bucket size parameter, we get same recovery time to recover k data buckets. Indeed, the time

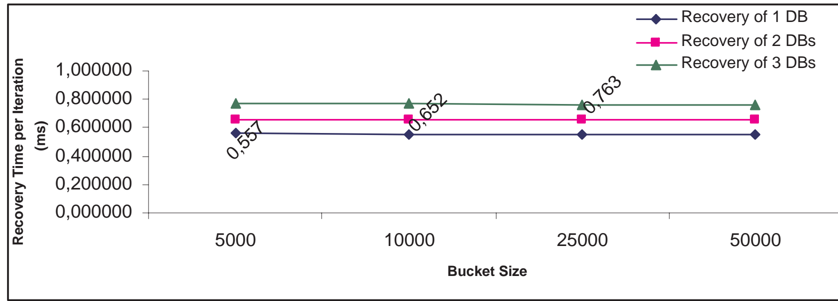


Figure 6: Performance Measurements of the recovery time per iteration, for k Data Buckets Recovery, $k \in \{1,2,3\}$, for bucket size $\in \{5000,10000,25000,50000\}$, and using UDP protocol.

to recover k data records per iteration is approximately 0.557 ms for $k = 1$, 0.652 ms for $k = 2$, and 0.763 ms for $k = 3$. The obtained results show that independently of the value of bucket size, there is constant time needed to compute a record per iteration, which is approximately equal to 0.10 ms for a record size set to 100 bytes. We notice that the time to recover a record decreases when k increases. This is due to the querying messages being factorized. Since, in each iteration, we query m-1 buckets, to recover k data records.

Despite the scalability and the linearity in the results obtained for different buckets sizes, we estimate that the UDP-based recovery scenario becomes unsuitable for higher values of bucket size. Thus, we have sketched a new scenario based on TCP/IP.

7.2 TCP/IP-based Data Buckets' recovery Scenario

At each iteration, the recovery manager asks participating buckets to search a *slice* of s records, corresponding to ranks: $r, \ldots, r+s-1$. Then, r is incremented by s. The buckets reply in the limit of the number of records they hold. And, on the receipt of the buffers, data records having same rank are retrieved from the buffers and from the local data structure, to compute missing records. Finally, the recovery manager sends to each spare data bucket its slice of recovered contents. All the buffers are sent through a TCP/IP-connection.

To measure the performance results of our scenario, we create a k-available LH*$_{RS}$ £le, k = 1,2,3. The created £le spreads over four data buckets. At each experiment, we simulate the failure of k data buckets, and then we recover them. We have varied the slice size, which is the number of data records recovered per iteration, to determine best performance. Table 3 reports on the performance results of recovering k data buckets, $k \in \{1, 2, 3\}$, using TCP/IP-based recovery

scenario.

Recovery of 1 Data Bucket

Slice	*Tot. Time (sec)*	*Com. Time (sec)*	*Process Time (sec)*
1250	50.352	46.968	1.161
3125	23.073	20.008	1.052
6250	13.650	10.523	1.043
15625	8.292	1.986	1.042
31250	6.700	3.134	1.042

Recovery of 2 Data Buckets

Slice	*Tot. Time (sec)*	*Com. Time (sec)*	*Process Time (sec)*
1250	62.620	57.083	2.495
3125	29.893	24.081	2.403
6250	19.529	14.243	2.402
15625	14.510	8.556	2.364
31250	12.718	7.580	2.354

Recovery of 3 Data Buckets

Slice	*Tot. Time (sec)*	*Com. Time (sec)*	*Process Time (sec)*
1250	82.619	74.227	3.996
3125	38.675	30.824	3.815
6250	26.028	18.387	3.775
15625	19.638	11.436	3.756
31250	17.825	9.233	3.735

Table 3: Performance Measurements of k Data Buckets Recovery, $k \in \{1,2,3\}$ using TCP/IP protocol.

7.3 Performance comparison

The cost of recovery of 31250 records using UDP lasts 17.375 sec. We get better performance results using TCP/IP when slices are either a £fth of the bucket: 13.65sec, a half: 8.292sec, or the entire bucket: 6.7sec. The best result is when we recover the entire bucket. The cost of recovery using UDP is 2.6 times greater than TCP/IP, so the improvement is 260%. Similar remarks hold for multiple data buckets recovery. Indeed, we get better performance results for slice in the range $[b/5, b]$, where b is the bucket size.

8 Conclusion

We have evaluated the LH*$_{RS}$ £le creation, parity bucket creation, data retrieval in both normal mode and degraded mode, and £nally the recovery of more than one data bucket. Experiments prove the ef£ciency of the proposed scenarios. They also prove the scalability of the LH*$_{RS}$ scheme. The curves of the total insert time are about linear function of the £le size. Hence should remain so for further inserts and £les scaling to much larger sizes. Likewise, the record processing times of a normal search or of a recovery remain constant. They should remain so for much further scaling £les.
Further work concerns in depth performance analysis and implementation improvements to our parity calculus [S02]. It will also concern the full integration of our system under SDDS-2002 prototype. Finally, we plan to investigate other encoding and decoding techniques.

References

[B00] F. Bennour. Un Gestionnaire de Structures Distribuées et Scalables pour les multiordinateurs Windows: Fragmentation par Hachage. *PhD thesis in French, Paris Dauphine University*, 2000.

[BB94] M. Blaum, J. Brady, J. Bruck & J. Menon. EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures. *IEEE*, 1994.

[BK95] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby & D.Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme. *ICSI Tech Rep.*,TR-95-048, 1995.

[H94] L. Hellerstein, G.A. Gibson, R.M. Karp, R.H. Katz & D.A.Patterson. Coding Techniques for handling Failures in Large Disk Arrays. *Algorithmica*,1994, 12, pp.182-208.

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

[LS00] W. Litwin & T.J.E. Schwarz. LH*$_{RS}$ A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. *Proceedings of the ACM SIGMOD* ,p.237-248, 2000.

[P97] J. S. Plank. A Tutorial on Reed-Solomon Coding for fault-Tolerance in RAID-like Systems. *Software – Practise & Experience 27(9)*,pp 995-1012,Sept. 1997.

[R89] M. O. Rabin. Ef£cient Dispersal of Information for Security, Load Balancing and Fault Tolerance. *Journal of ACM, Vol. 26*,pp. 335-348, April 1989.

[S02] T.J.E. Schwarz. Generalized Reed Solomon code for erasure correction. *4-th International Workshop on Distributed Data and Structures*, March 2002.

[SDDS] http://ceria.dauphine.fr/SDDS-bibliographie.html.

[XB99] L. Xu & J. Bruck. Highly Available Distributed Storage Systems. *Proceedings of workshop on Distributed High Performance Computing, Lecture notes in Control and Information Sciences, Springer Verlag*,1999.

Rim Moussa is a PhD student with CERIA Lab., Dauphine University, Pl. du Mal de Lattre, Paris 75016, France. E-mail: Rim.Moussa@dauphine.fr

Witold Litwin is the director of CERIA Lab., and professor at Dauphine University, Pl. du Mal de Lattre, Paris 75016, France. E-mail: Witold.Litwin@dauphine.fr

Acknowledgements We thank Prof. Thomas Schwarz and Dr. Jim Gray for helpful discussions and comments. This work was partly supported by the research grants from Microsoft Research, and by the European Commission project ICONS project no. IST-2001-32429 as well as by the scholarship from Tunisian government.

d'adressage adéquates, tout en exploitant le multicast pour créer de nouvelles cases: de données, de parité ou de secours.

- La directive *FAST* définie dans le projet *client*, permet une meilleure gestion des accusés par l'envoi stipulant une fenêtre d'émission coulissante.
- Les directives *AckToPBs* et *AckToDB*, définies respectivement dans les projets *serveur de données* et *serveur de parité*, assurent le contrôle de flux et l'acquittement de messages entre les *serveurs de données* et les *serveurs de parité*.

ANNEXE B: PROTOTYPE USER GUIDE

1. Introduction

The present document is a guide to the test of the main functionalities of the LH*$_{RS}$ 2004 prototype. Please notice that our code is written in the aim of validation of our architectural and algorithmic choices through performance measurements, so the prototype doesn't deal with some exceptions (as program input error or any command mishandling etc…).

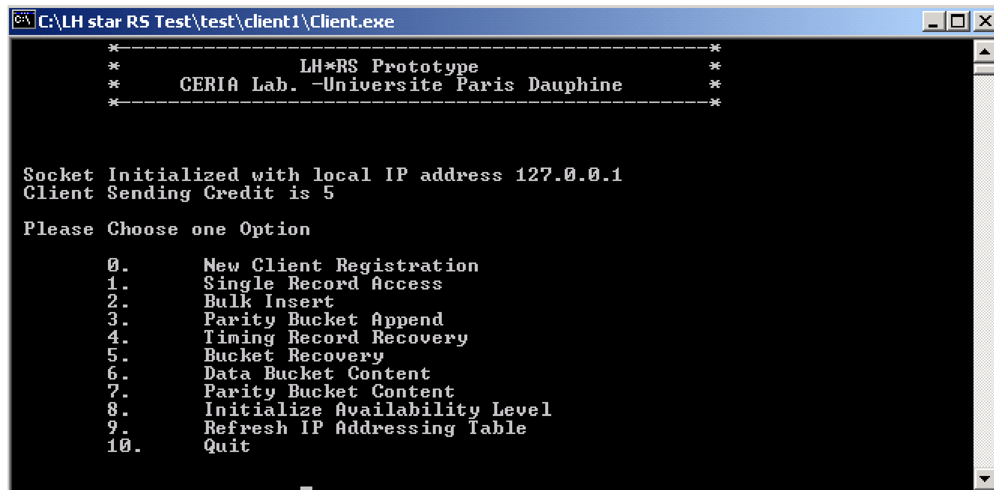
We have four types of executable files, that correspond in the LH*$_{RS}$ scheme to the *Client*, the *Coordinator*, the *Data Bucket* and the *Parity Bucket*. You are free to launch the . *exe files* on any machine.

All *Data Buckets* are connected to the Multicast IP address *233.1.1.1*, while *Parity Buckets* are connected to the Multicast IP address *233.1.1.2*. The buckets can be used for creating new buckets during data bucket split, or group high availability increase, as well as spare buckets. Notice also, that we can launch more than one data/ parity bucket on the same machine, one will be connected to its corresponding multicast group, while others wait for the completion of the first bucket disconnection from it's multicast group.

2. User-application Interface

Code	Option Title	Description
0	Client Registration	First of all, any *Client*, when launched listens temporarily to the UDP listening port 6060. It must subscribe to the *Coordinator* to get a definitive entity number and communicate with the *Coordinator* and the *Data Buckets*. In Option 0, the user types the IP address of the Coordinator.
1	Single Record Access	Option 1 allows the user to insert, update, delete or retrieve one record. In case of retrieve, and bucket failure, the

	Access	record is recovered.
2	Bulk Insert	Data Buckets have limited bucket sizes. The latter are displayed on any data bucket window. One has to think to the number of data buckets and the file size, before executing bulk inserts option (option 2).
3	Parity Bucket Append	Option 3 allows the increase of the availability of one bucket's group. We show on the application windows the interaction of each involved data bucket with the new created parity bucket, as well as total, communication and processing times.
4	Timing Record Recovery	Option 4 allows measuring the time to recover records belonging to a failed data bucket, on the parity bucket.
5	Bucket Recovery	Option 5 allows failure detection and recovery of all failed buckets of one group.
6	Display Data Bucket Content	Option 6 allows data bucket display. The user must specify the data bucket logical number.
7	Display Parity Bucket Content	Option 7 allows parity bucket display. The user must specify the parity bucket entity number and the IP address hosting the parity bucket.
8	Initialize Availability Level	Option 8 allows initialization of availability level of group 0, and even auto-creation of new added parity buckets just for group 0.
9	Client IP Address Table Refreshment	Option 9 allows a *Client* to refresh its *IP address Table* by asking the *Coordinator* to send to him back all the IP addressing and communication settings of all existing Data Buckets.



3. System Requirements

Our prototype is written in C and uses WinSockets 1.1. We tested our prototype on machines running Windows 2K server Operating System.

RAM requirements depend on the following parameters (1) the Maximal Bucket Size (corresponding to the pre-compiler directive *BucketSize* in Data/Parity Server project), which the server was initialised with; and (2) the Record Size (corresponding to the pre-compiler directive *RecordSize* in Data/Parity Server project). In addition, memory is allocated for buffering during bucket split, recovery etc…

In the demonstration version, the *Bucket Size* is set to 1000 records, and the *Record Size* is set to 108 bytes (non-key field is 100 bytes). Consequently, each server consumes at least 108 Kbytes.

PUBLICATIONS

Experimental Performance Analysis of LH*$_{RS}$ Parity Management

RIM MOUSSA
CERIA Lab.-Université Paris dauphine, France

WITOLD LITWIN
CERIA Lab.-Université Paris dauphine, France

Abstract

We present an implementation of LH*$_{RS}$: a high-availability Scalable Distributed Data Structure on Windows 2000. We show experimental determination of the time for £le creation, key search, record recovery, parity bucket creation, and data bucket recovery. The measures prove the ef£ciency of the LH*$_{RS}$ scheme, the scalability especially.

Keywords

High Availability, Fault-tolerance, Scalability, Distributed Data Structures, Reed Solomon codes, Multicomputers

1 Introduction

Scalable Distributed Data Structures [SDDS] are being developed for computers over fast networks, usually local networks, i.e. for the multicomputers. This new hardware architecture is promising and is becoming highly popular. In spite of the advantages given by the data distribution layout, failures remain the problem, and the vulnerability accentuates with the increase of the number of machines in the network. Many approaches to build highly available, i.e., fault tolerating, distributed data storage systems have been proposed. They generally fall into the two categories of (i) data mirroring and (ii) parity information. The latter approach uses erasure correcting codes to guard against failures The simplest codes, e.g. in RAID systems, use XOR calculus for the tolerance of a single site failure. For multiple failures more complex codes are needed. These can be the binary codes [H94] for double or triple failure, or character codes. Examples of character codes are: the array codes as the EVENODD code [BB94], the X-code [XB99] or the Reed Solomon codes. The latter appear best at present to deal with multiple failures [R89] [BK95] [P97] [LS00].

The Scalable and Distributed Data Structure (SDDS) schema called [LS00] provides high-availability server nodes of data buckets by using Reed Solomon codes. We present our contribution to the design and implementation of a prototype system for LH*$_{RS}$ £les on Windows 2000 multicomputers. The goal of the prototype is to tune the system and experimentally determine the LH*$_{RS}$ performance. Our LH*$_{RS}$ implementation completes and improves the one presented in [L00]. The entire prototype reuses an existing LH* implementation that does not provide the high-availability [B00]. Our contribution to the prototype design consists in the solutions to the problems encountered by the £rst prototype, namely, (1) a more reliable data bucket split, (2) a more ef£cient data buckets'recovery, and (3) a parity bucket creation algorithm increasing the availability of a data bucket group. We show experimental determination of the time for £le creation, key search, record recovery, parity bucket creation, and data bucket recovery.

Below, Section 2 recalls the LH*$_{RS}$ £le structure. Section 3 overviews our contribution to the implementation. Further sections present the performance analysis. Section 8 concludes the article.

The hardware testbed consists of six machines; each has 128 MB of RAM, with a 731MHz Pentium III processor, and runs Windows 2000. All the machines are connected by a 100 Mbps Ethernet. For the experiments, the record size is set to 100 bytes and the con£guration tested consists of one client, four data buckets and k parity buckets; $k \in \{0, 1, 2\}$.

2 LH*$_{RS}$ Data Structure

LH*$_{RS}$ scheme is described in detail in [LS00]. An LH*$_{RS}$ £le is subdivided into groups. Each group is composed of m data buckets and k parity buckets. The *data buckets* store the data records, while the parity buckets store the parity ones. Every data record £lls a rank r in its data bucket. A record group consists of all records with the same rank in a bucket group. We construct parity records from data records having the same rank within data buckets forming a bucket group Fig. 1a. The record grouping has an impact on the data structure of a parity record. The latter keeps track of the data records it is computed from. Fig. 1b-c, shows each the structure of a data record and a parity record. The parity calculus is done using Reed Solomon codes.

The £le starts with one data bucket and one parity bucket. It scales up through data buckets splits, as the data buckets get overloaded. Each data bucket contains a maximum number of b records. The value of b is the bucket capacity. When the number of records within a data bucket exceeds b, the bucket adverts a special entity called the split coordinator. The latter designates a data bucket to split.

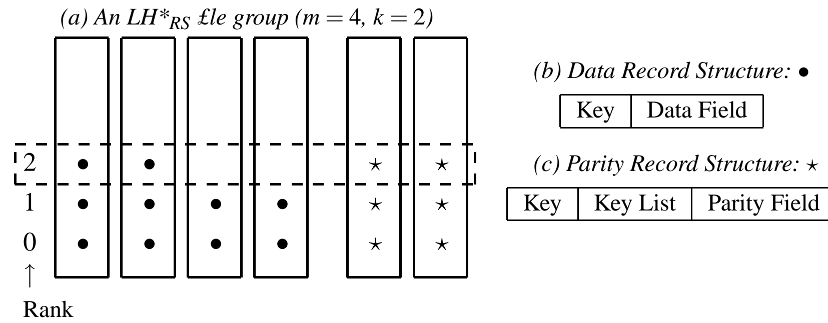


Figure 1: An LH*$_{RS}$ £le Structure.

3 File Creation

3.1 Bucket Splitting

During a data bucket split, half of the splitting data bucket move to a newly created data bucket. As a consequence to the data transfer, the records remaining in the data bucket and those moving get new ranks. The parity buckets belonging to $g1$, the splitting data bucket group, and to $g2$, the new data bucket's group, have to be updated. A £rst update scenario is proposed in [L00]. It consists in sending update messages to parity buckets. Each parity bucket belonging to $g1$ receives one message per record to delete, i.e., b delete messages in total and one message per record to insert, i.e., about $b/2$ insert messages, a total of $1.5*b$ update messages. Same for each parity bucket belonging to $g2$, it receives $b/2$ insert messages. Those messages are sent through an unreliable protocol that is UDP without acknowledgements. The parity buckets may be inconsistent if the messages are lost, besides the communication is inef£cient since many small messages are sent.

We have implemented a splitting scenario overcoming disadvantages of the above one. We have replaced UDP by TCP/IP, sending a single message with all parity records to update. The update buffer is a collection of the structure S:

- Record's key;
- Record's Data £eld;
- Old Rank: the rank occupied by the record; and it has to be deleted from it;
- New rank: the rank of the record after the split, that it has to be inserted on it.

We now give the details of this scenario at the level of the involved buckets:

→*The splitting data bucket:*
The splitting data bucket £lls in each structure S as follows: If the data record is moving: then it is to be deleted from parity records of rank *oldrank* in the parity buckets, belonging to the same group as

the splitting data bucket. And, *newrank* is set to -1. Else, if the data record is staying: the data record is to be £rst deleted from parity records of rank $oldrank$, and then reinserted at rank *newrank*.

→*New data Bucket:*
Following a data bucket split, the new data bucket receives a buffer, containing about $b/2$ records to insert. While inserting, it prepares a buffer to update the content of its parity buckets. The parity bucket uses the structure S, the £eld $oldrank$ is set to –1, and the £eld *newrank* indicates the rank occupied by the record just inserted.

→*Parity bucket:*
Each parity bucket, receiving an update buffer, processes each structure S as follows: (1)If $S.oldrank \neq 1$, delete the data record $\{S.key$, $S.datafield\}$ from parity record of rank $oldrank$. (2) Else if $S.newrank \neq 1$, insert the data record $\{S.key$, $S.datafield\}$ in the parity record of rank *newrank*.

3.2 Experimentation description

First we create a £le of 25000 records. The £le spread on four data buckets, each containing 6250 records. During the £le creation, three splits occurred. The £rst split is of the data bucket numbered 0. Then, data buckets 0 and 1 split at approximately the same time. Hereafter, Figure 2 reports on £le creation performance.

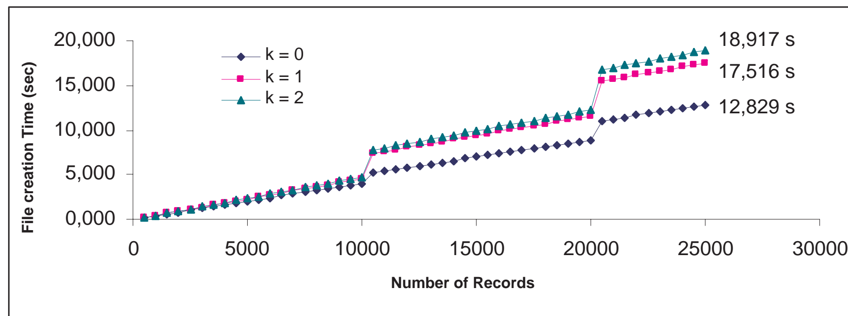


Figure 2: Performance Measurements of the creation of a k-available LH*$_{RS}$ File, that spread on 4 data buckets (25000 data records) k $\in \{0, 1, 2\}$

Following a data bucket split, there is an increase in the insert average time of 40% from $k = 0$ to $k = 1$, and of 7% from $k = 1$ to $k = 2$. The increase is mainly due to the preparation and sending of TCP/IP update buffers (case of $k \geqslant 1$) from the splitting data bucket and new data bucket to their group reliability. We

estimate that the splitting process may become cumbersome when k increases, and a more ef£cient parity buckets' update strategy is then required. One can release the splitting data bucket and the new data bucket from the task of communicating the updates to all parity buckets of its group.

To prove the scalability of the £le creation scenario, we conduct experiments £rst on the creation of an LH*$_{RS}$ £le, that spread on 8 data buckets, and containing 50000 records (see Figure 3), and another £le containing 25000 records, that spread on 4 data buckets, but has group size 2. We obtain an insert time per record approximately equal to {0.40ms, 0.44ms, 0.48ms} respectively to $k \in \{0, 1, 2\}$. We notice that the insert time per record is invariant of group size and of the number of records inserted.

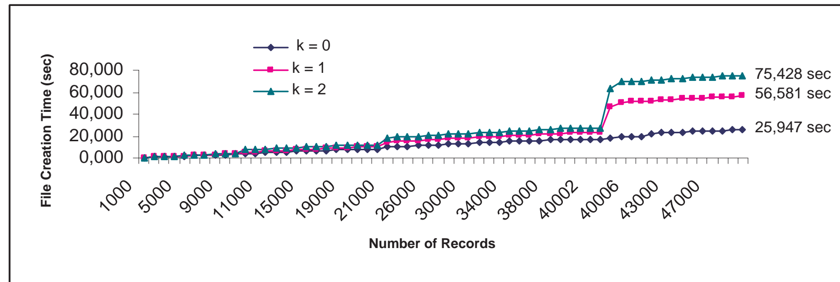


Figure 3: Performance Measurements of the creation of a k-available LH*$_{RS}$ File, that spread on 8 data buckets (50000 data records) k $\in$ {0, 1, 2}

4 Parity Bucket Creation

The high availability scenario ensures the increase of the availability of a group, just by adding a parity bucket. We begin by describing the scenario, then we on report the time to create a parity bucket.

4.1 Creation Scenario

The new parity bucket is created in at most $x \leq m$ steps, such that x is the number of not dummy data buckets in the group g. At each step, the parity bucket establishes a TCP/IP connection with one data bucket, and the latter sends its contents. The received buffer is so processed for insert/ update.

Each data bucket of the group adds the new parity bucket to the list of its group reliability, and will be able to re¤ect the client's manipulations on this parity bucket.

4.2 Parity Bucket Creation Performances

In each experiment, we create a £le of 2.5∗Bucket Size records. The £le spreads on up to four data buckets, each of 0.625∗Bucket Size records. The performance results are in Table 1. We highlight *The Communication Time*, i.e. the time spent on establishing TCP/IP connections to receive each data bucket content; and *The Process Time*, i.e. time spent on processing the buffer.

BucketSize	*Com.Time(sec)*	*ProcessTime(sec)*	*Total(sec)*
5000	2,211	0,302	2,523
10000	2,433	0,611	3,044
25000	3,185	1,652	4,847
50000	4,667	3,395	8,062

Table 1: Parity Bucket Creation Performance Results

We notice that the time spent on establishing TCP/IP connections with the four data buckets is relatively high compared to process time. Indeed, the communication time is {87.63%, 79.93%, 65.71%, 57.89%} for buckets sizes {5000, 10000, 25000, 50000}. It appears interesting to choose a larger bucket to amortize TCP/IP connection cost.

5 Basic Key Search Performance

We measure both parallel searches, during which a client sends a ¤ow of search messages in parallel to the four data buckets, and synchronized search, where the client waits for a reply before issuing another request. Figure 4 presents the times for a £le of size 125000 distributed over 4 buckets.

The search times per record are essentially independent of the number of searches. The synchronized search time measures the server access time. Let's notice that it is 30 times faster than that to a £le on a typical local disk. The parallel search time measures the maximal client speed. Notice also that this search time is about 100 times faster than that to a local disk.

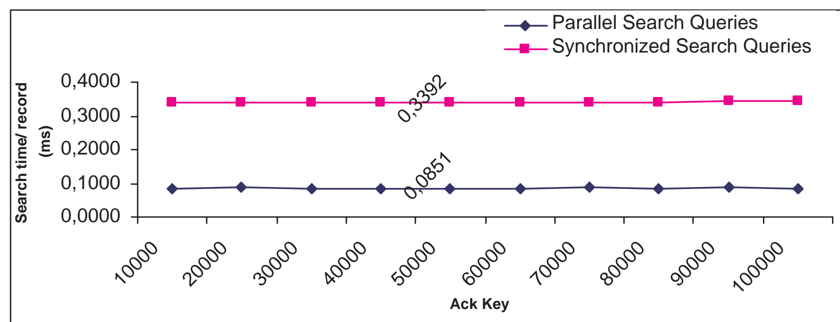


Figure 4: Search Performance Results

6 Data Record Recovery

6.1 Recovery Algorithm

The routine "recover one record" follows the following steps:

1. Look for the data record's key inside the parity bucket structure,
2. Send search queries to alive buckets,
3. Wait until receiving replies to sent queries,
4. Compute the missing data record through RS-decoding,
5. Send the recovered record to the client.

6.2 Performance Analysis

We create an LH*$_{RS}$, 1-available £le of 125000 records. Then, we simulate the failure of one data bucket. Figure 5 summarizes the performance results of recovering x records of the failed data bucket.

The time to look for a data record's key inside a parity bucket structure, of 32150 records, is approximately equal to $0.7ms$. It is 56% of the average time to recover a data record.
Notice that while the record recovery time is 15 times higher than that of a normal key search, it is still 8 times faster than a normal key search on a local disk.

7 Data Bucket Recovery

We describe and we report the performance results of two scenarios devised.

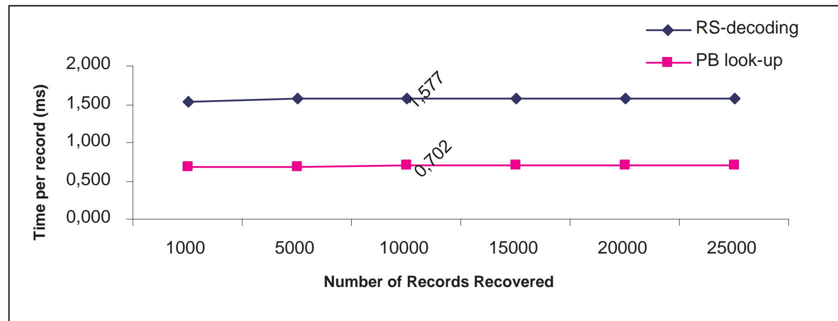


Figure 5: Data Records Recovery Performance Results

7.1 UDP-based Data Buckets' recovery Scenario

The data buckets' recovery scenario starts at the coordinator level. The coordinator probes all buckets belonging to the group, waits a time-out, then either noti£es the application of the impossibility of recovery, due to the lack of surviving buckets; or assigns to the £rst parity bucket replying to the probe the task of failed buckets' recovery. In the case of recovery, the elected parity bucket is adverted of the bucket's states and addresses. It chooses m buckets among surviving ones, parity buckets being preferred. This is to let the data buckets being available for the application requests, while the recovery is in progress.

To measure the performance of the scenario, we create a k-available LH*$_{RS}$ £les, with $k \in \{1, 2, 3\}$. The created £le spreads over four data buckets. At each experiment, we simulate the failure of k data buckets, and then we recover them. Table 2 reports decoding performance for the recovery of k data buckets, such that k $\in \{1, 2, 3\}$ for different bucket sizes.

Bucket Size	Bucket contents	1 DB recovery (s)	2 DBs recovery (s)	3 DBs recovery (s)
5000	3125	1.742	2.043	2.564
10000	6250	3.465	4.076	4.797
25000	15625	8.662	10.185	11.928
50000	31250	17.375	20.349	23.895

Table 2: Performance Measurements of k Data Buckets Recovery using UDP protocol, $k \in \{1,2,3\}$.

Figure 6 proves the scalability of the algorithm. Independently of bucket size parameter, we get same recovery time to recover k data buckets. Indeed, the time

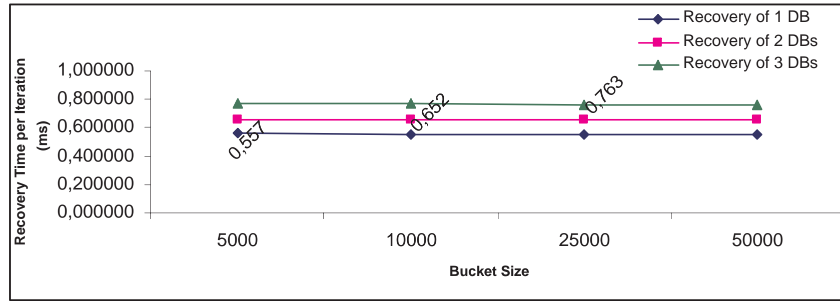


Figure 6: Performance Measurements of the recovery time per iteration, for k Data Buckets Recovery, $k \in \{1,2,3\}$, for bucket size $\in \{5000, 10000, 25000, 50000\}$, and using UDP protocol.

to recover k data records per iteration is approximately 0.557 ms for $k = 1$, 0.652 ms for $k = 2$, and 0.763 ms for $k = 3$. The obtained results show that independently of the value of bucket size, there is constant time needed to compute a record per iteration, which is approximately equal to 0.10 ms for a record size set to 100 bytes. We notice that the time to recover a record decreases when k increases. This is due to the querying messages being factorized. Since, in each iteration, we query m-1 buckets, to recover k data records.

Despite the scalability and the linearity in the results obtained for different buckets sizes, we estimate that the UDP-based recovery scenario becomes unsuitable for higher values of bucket size. Thus, we have sketched a new scenario based on TCP/IP.

7.2 TCP/IP-based Data Buckets' recovery Scenario

At each iteration, the recovery manager asks participating buckets to search a *slice* of s records, corresponding to ranks: $r, \ldots, r+s-1$. Then, r is incremented by s. The buckets reply in the limit of the number of records they hold. And, on the receipt of the buffers, data records having same rank are retrieved from the buffers and from the local data structure, to compute missing records. Finally, the recovery manager sends to each spare data bucket its slice of recovered contents. All the buffers are sent through a TCP/IP-connection.

To measure the performance results of our scenario, we create a k-available LH*$_{RS}$ £le, k = 1,2,3. The created £le spreads over four data buckets. At each experiment, we simulate the failure of k data buckets, and then we recover them. We have varied the slice size, which is the number of data records recovered per iteration, to determine best performance. Table 3 reports on the performance results of recovering k data buckets, $k \in \{1, 2, 3\}$, using TCP/IP-based recovery

scenario.

Recovery of 1 Data Bucket

Slice	*Tot. Time (sec)*	*Com. Time (sec)*	*Process Time (sec)*
1250	50.352	46.968	1.161
3125	23.073	20.008	1.052
6250	13.650	10.523	1.043
15625	8.292	1.986	1.042
31250	6.700	3.134	1.042

Recovery of 2 Data Buckets

Slice	*Tot. Time (sec)*	*Com. Time (sec)*	*Process Time (sec)*
1250	62.620	57.083	2.495
3125	29.893	24.081	2.403
6250	19.529	14.243	2.402
15625	14.510	8.556	2.364
31250	12.718	7.580	2.354

Recovery of 3 Data Buckets

Slice	*Tot. Time (sec)*	*Com. Time (sec)*	*Process Time (sec)*
1250	82.619	74.227	3.996
3125	38.675	30.824	3.815
6250	26.028	18.387	3.775
15625	19.638	11.436	3.756
31250	17.825	9.233	3.735

Table 3: Performance Measurements of k Data Buckets Recovery, $k \in \{1,2,3\}$ using TCP/IP protocol.

7.3 Performance comparison

The cost of recovery of 31250 records using UDP lasts 17.375 sec. We get better performance results using TCP/IP when slices are either a £fth of the bucket: 13.65sec, a half: 8.292sec, or the entire bucket: 6.7sec. The best result is when we recover the entire bucket. The cost of recovery using UDP is 2.6 times greater than TCP/IP, so the improvement is 260%. Similar remarks hold for multiple data buckets recovery. Indeed, we get better performance results for slice in the range $[b/5, b]$, where b is the bucket size.

8 Conclusion

We have evaluated the LH*$_{RS}$ £le creation, parity bucket creation, data retrieval in both normal mode and degraded mode, and £nally the recovery of more than one data bucket. Experiments prove the ef£ciency of the proposed scenarios. They also prove the scalability of the LH*$_{RS}$ scheme. The curves of the total insert time are about linear function of the £le size. Hence should remain so for further inserts and £les scaling to much larger sizes. Likewise, the record processing times of a normal search or of a recovery remain constant. They should remain so for much further scaling £les.
Further work concerns in depth performance analysis and implementation improvements to our parity calculus [S02]. It will also concern the full integration of our system under SDDS-2002 prototype. Finally, we plan to investigate other encoding and decoding techniques.

References

[B00] F. Bennour. Un Gestionnaire de Structures Distribuées et Scalables pour les multiordinateurs Windows: Fragmentation par Hachage. *PhD thesis in French, Paris Dauphine University*, 2000.

[BB94] M. Blaum, J. Brady, J. Bruck & J. Menon. EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures. *IEEE*, 1994.

[BK95] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby & D.Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme. *ICSI Tech Rep.*,TR-95-048, 1995.

[H94] L. Hellerstein, G.A. Gibson, R.M. Karp, R.H. Katz & D.A.Patterson. Coding Techniques for handling Failures in Large Disk Arrays. *Algorithmica*,1994, 12, pp.182-208.

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

[LS00] W. Litwin & T.J.E. Schwarz. LH*$_{RS}$ A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. *Proceedings of the ACM SIGMOD* ,p.237-248, 2000.

[P97] J. S. Plank. A Tutorial on Reed-Solomon Coding for fault-Tolerance in RAID-like Systems. *Software – Practise & Experience 27(9)*,pp 995-1012,Sept. 1997.

[R89] M. O. Rabin. Ef£cient Dispersal of Information for Security, Load Balancing and Fault Tolerance. *Journal of ACM, Vol. 26*,pp. 335-348, April 1989.

[S02] T.J.E. Schwarz. Generalized Reed Solomon code for erasure correction. *4-th International Workshop on Distributed Data and Structures*, March 2002.

[SDDS] http://ceria.dauphine.fr/SDDS-bibliographie.html.

[XB99] L. Xu & J. Bruck. Highly Available Distributed Storage Systems. *Proceedings of workshop on Distributed High Performance Computing, Lecture notes in Control and Information Sciences, Springer Verlag*,1999.

Rim Moussa is a PhD student with CERIA Lab., Dauphine University, Pl. du Mal de Lattre, Paris 75016, France. E-mail: Rim.Moussa@dauphine.fr

Witold Litwin is the director of CERIA Lab., and professor at Dauphine University, Pl. du Mal de Lattre, Paris 75016, France. E-mail: Witold.Litwin@dauphine.fr

Acknowledgements We thank Prof. Thomas Schwarz and Dr. Jim Gray for helpful discussions and comments. This work was partly supported by the research grants from Microsoft Research, and by the European Commission project ICONS project no. IST-2001-32429 as well as by the scholarship from Tunisian government.

Design and Implementation of LH*$_{RS}$
Highly available Data Storage Syster

RIM MOUSSA
CERIA Lab.-Université Paris dauphine, France

THOMAS J.E. SCHWARZ, S.J.
Santa Clara University, USA

Abstract

The ideal storage system is always available and is incrementally expar able. Existing storage systems are far from this ideal. Affordable comput and high-speed networks allow us to investigate storage architectures t bring us closer to the ideal storage system. We describe a prototype imp mentation of the highly available scalable distributed data structure LH*. The scheme allows to recover from a multiple unavailability using a vari of Reed Solomon erasure correcting code. We present the system architect and experimental performance measurements.

Keywords

High availability, Scalable and Distributed Data Structures, Reed Solomon C Erasure-resilient systems

1 Introduction

The Scalable and Distributed Data Structures [SDDS] are being devel computers over fast networks, usually local networks, i.e. for the multico This new hardware architecture is promising and becomes highly popula of the advantages given by the data distribution layout, vulnerability to fa mains the arena, and accentuates with the increase of the number of ma the network. Many approaches to build highly available, i.e., fault tolera tributed data storage systems have been proposed. They generally fall two categories of (i) data mirroring, and (ii) parity information. In th approach the storage overhead is prohibitive. The latter approach uses correcting codes to guard against failures. The simplest codes, e.g. in R tems [PGK88], use XOR calculus for the tolerance of a single site fai multiple failures more complex codes are needed. These can be the bin [H94] for double or triple failure, or character codes. Examples of charac

are: the array codes as the EVENODD code [BB94], the X-code [XB9
Reed Solomon codes. The latter appear best at present to deal with mult
ures [R89, BK95, P97, LS00, M00, ML02]. Theoretical proofs demonstr
periority of erasure resilient systems to replicated systems can be found
WK02].

Below, Section 2 recalls the LH*RS file structure. Section 3 overvie
posed architecture for LH*$_{RS}$. Performance results are given in section 4
section 5 concludes the article.

2 LH*$_{RS}$ Scheme

LH*$_{RS}$ scheme is described with details in [LS00, S02, ML02]. An LH*
subdivided into groups. Each group is composed of *m* Data Buckets a
ity Buckets. The data buckets store the data records, same for parity
Every data record fills a rank *r* in its data bucket. A record group consi
records with the same rank in a bucket group. We construct parity reco
data records having the same rank within data buckets forming a buck
(see Fig. 1(a)). The record grouping has an impact on the data structure
ity record. The latter keeps track of the data records it is computed fr
1(b-c) shows each of the structure of a data record and a parity record. T
calculus is done using Reed Solomon codes.

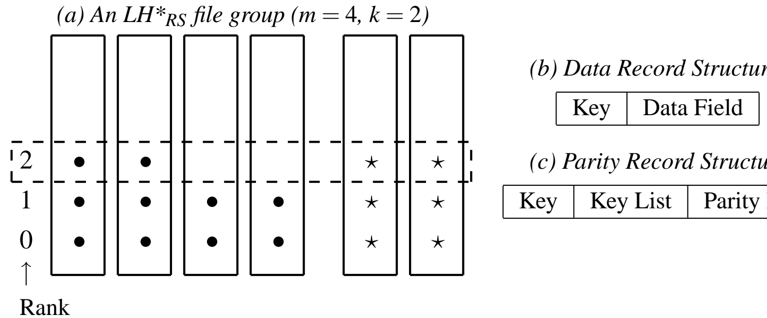


Figure 1: An LH*$_{RS}$ file Structure.

In the scenarios described below, all the buffers are sent through TC
performance and reliability concerns demonstrated in [M00, ML02], w
compared TCP/IP-based scenarios to UDP-based scenarios.

2.1 Data Bucket Splitting Scenario

The file starts with one data bucket and *k* parity bucket. It scales up thro
buckets' splits, as the data buckets get overloaded. Each data bucket c

maximum number of b records. The value of b is the bucket capacity. V
number of records within a data bucket exceeds b, the bucket adverts a
entity coordinating the splits, which is the coordinator. The latter desi
data bucket to split. During a data bucket split process, half of the split
bucket contents move to a new created data bucket. As a consequence to
records' transfer, the data records remaining in the splitting data bucket a
moving get new ranks. So, the parity buckets belonging to (i) $g1$: the
data bucket's group, and (ii) $g2$: the new data bucket's group, have to be
In that way during the split process, two update buffers are filled respec
the splitting data bucket and the new data bucket, and sent to update th
buckets of each group.

2.2 High Availability Scenario

The high availability scenario ensures the increase of the availability of
just by adding a parity bucket. The new parity bucket executes at mos
stages, such that x is the number of not dummy data buckets in the gro
each stage, the parity bucket updates its contents with respect to each dat
contents. Each data bucket of the group adds the new parity bucket to th
its group reliability, and will be able to reflect the client's manipulation
parity bucket.

2.3 Bucket Recovery Scenario

The data buckets' recovery scenario starts at the coordinator level. The c
tor probes all buckets belonging to the group, waits a time-out, then eithe
the application of the impossibility of doing recovery, due to the lack o
ing buckets; or assigns to the first parity bucket replying to the probe th
failed buckets' recovery. In case of possible recovery, the elected parity b
adverted of the bucket's states and addresses. It chooses m buckets am
viving one's, in preference parity buckets. That is to let the data bucke
to application requests. At each iteration, the recovery manager asks part
buckets to search slice records, corresponding to ranks in range [r, , r+
Then, r is incremented of slice. The buckets reply in the limit of the nu
records they hold. And, on the receipt of the buffers, data records havi
rank are retrieved from the buffers and from the local data structure, to
missing records. Finally, the recovery manager sends to each spare data b
partial recovered contents.

3 System Architecture

In [M00][ML02], we have implemented our scenarios related to file creat
bucket split, high availability and buckets recovery, on the top of SDDS
chitecture [D01]. SDDS2000 architecture was proposed by F. Bennour a
Dine, as an architecture for LH* and RP*. In order to have better perf
results, we embedded to SDDS2000 other components, namely: (i) an
TCP/IP connections handler, (ii) flow control and acknowledgements
and (iii) a dynamic addressing structure. To M. Welch and al. [WGBC0
fication of server architectures, our architecture is an hybrid architecture
it falls in the spectrum between multithreaded architectures and event-d
chitectures, since it combines multithreaded architectures and queues. Th
communicate through events and queues, and concurrent threads are sync
using common concurrency-programming tools.

3.1 TCP/IP Connections Handler

In our last implementation of LH*$_{RS}$ scenarios mapped to SDDS2000 arc
[ML02], the communication time dominates the total time; especially f
based scenarios, i.e., the parity bucket creation scenario and the bucke
ery scenario. To improve the performance results, we have enriched SD
architecture with an efficient TCP/IP connections handler, and mapped
narios to the new architecture. According to RFC 793 [ISI81] and [MI
can open TCP/IP connections in a passive OPEN way, i.e, a process wi
and queue incoming connection requests. The backlog parameter desig
number of pending TCP/IP connections. The Windows Sockets 1.1 spec
indicates that the maximum allowable value for a backlog is 5; however, V
2000 Server accepts a backlog of 200, and Windows 2000 Professiona
a backlog of 5. Figure 2 and Figure 3, illustrate the way we establish a
connection respectively in SDDS2000 and in the new devised archite
SDDS2000, in order to establish a TCP/IP connection, first two messa
through UDP are exchanged between the two peers. Added to that overh
delay underwent to establish the connection while each peer executes a
ate APIs. In our architecture, a TCP listening thread is instantiated on th
creation, and handles any incoming connection. Likewise, we don't nee
chronize the peers to establish a TCP/IP connection between them, sinc
sides TCP/IP connections are passive OPEN, and the *sender* peer execute
priate APIs, without asking the *receiver* peer to get ready.

3.2 Flow Control and Acknowledgement Strategy

Diéne [D01] proposed a flow control and acknowledgement strategy, to
messages losses under UDP protocol. With respect to his strategy, we

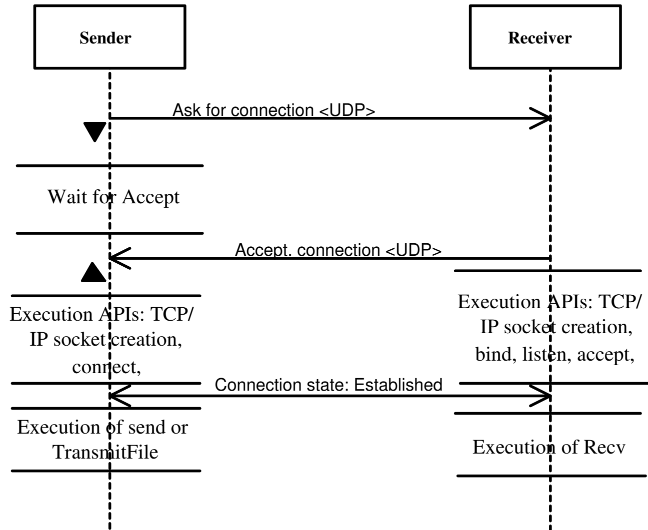


Figure 2: TCP/IP Connection in SDDS2000.

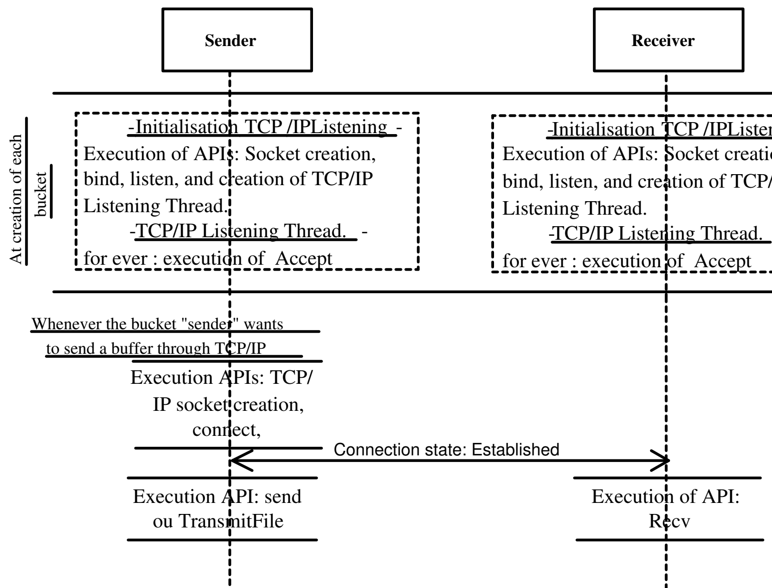


Figure 3: TCP/IP Connections Handler.

our flow control and acknowledgement strategy that deploys only one a
thread, while Diéne's strategy deploys *Window Size*+2 threads. The *Win*
parameter is the number of messages that could be sent without ackno
ments, such that each thread handles one message at a time. For that purp
peer has a *Sending Credit* (or Window size), that when it reaches zero, th
stops sending messages, else the sending process pulls a free position fro
managed *Free Positions Queue*, adds the message to *Not Yet Acquitted M*
List. The message is obviously removed when the corresponding ackno
ment is received; consequently a new position is signaled free, and q
Free Positions Queue. The *Acknowledgement Manager Thread* scans per
the list, checks sending time of each message, and re-sends if necessary
sage whenever the maximum number of re-sends is not exceeded. In the
the message is removed, and two cases are considered. Indeed, either th
peer commits an addressing error or the *receiver* peer is failed. In all c
coordinator is informed.

3.3 Dynamic Addressing Structure

In our first implementation, a static table containing the IP addresses o
ferent involved peers: clients, data/ parity buckets, is used for addressing
We proposed a new scenario to add a data/parity bucket to the file on-lin
the addressing table evolves accordingly to the file and is not user-fixed.
purpose, a bucket depending on its type data or parity bucket, is either c
to the *Data Buckets Multicast Group* or the *Parity Buckets Multicast G*
starts with the *Multicast Listening Thread* and the *Multicast Working*
The former listens to a fixed *Data /Parity Multicast Port*, and queues
messages, and the latter processes queued multicast messages. When th
receives a multicast message inviting the bucket to be a new or a spar
it instantiates the other threads, responds positively to the coordinator, a
for the confirmation. A selected bucket, upon confirmation receipt, dis
from its multicast group, while the non-selected buckets cancel the ins
process, and can commit to other invitations. The new bucket selection
has without doubt changed our architecture, and is applied to each of da
split scenario, high availability scenario and bucket recovery.

3.4 Architecture

Hereafter, we briefly describe each functional thread, the overall bucket
ture is illustrated in Figure 4.

> ↬ ***Multicast Listening Thread:*** It is a temporary thread. The thre
> listens to a fixed data or parity multicast port, and queues multic
> messages.

↬ ***Multicast Working Thread:*** It is also a temporary thread. T
thread processes queued multicast messages.

↬ ***UDP Listening Thread:*** It listens to a fixed UDP port. The latt
is deduced from the bucket number.

↬ ***Pool of Working Threads:*** A working thread processes queu
UDP messages and internal tasks.

↬ ***TCP Listening Thread:*** It accepts and handles multiple TCP/
connections.

↬ ***Acknowledgement Manager Thread:*** It scans the *Not Yet Acqu
ted Messages List*, checks sending time of each message, and r
sends if necessary the message whenever the maximum number
re-sends is not exceeded, otherwise deletes the message.

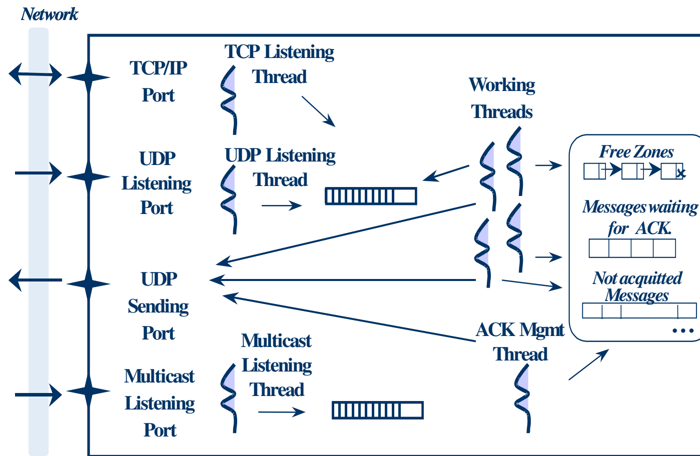


Figure 4: System Architecture.

4 Performance Results

The goal of the prototype is to tune and experimentally determine the LH
formance characteristics. Our LH*$_{RS}$ implementation improves that pres

[ML02]. It uses distributed RAM memory, and includes a data and parity
manager, and a basic query manager. Other functionalities are implem
key-based search queries, data update queries and propagation of update
ity buckets, record's recovery using UDP, buckets' recovery through UDP
bucket contents and statistics, etc.

The hardware testbed consists of six machines; each one has 512 MB
with a 1.8GHz Pentium processor, and runs Windows 2K Server. All the r
are connected to a regular Ethernet configuration with a max bandwidth o
For the experimental set up, the *Record Size* is set to 100 bytes and th
Size is set to 4 buckets. Performance results are expected to degrade fo
values of *Record Size* and *Group Size*. The best obtained performance re
using Reed Solomon codes with encoding and decoding in Galois Field:
Experiments details and a full comparison between different architect
configurations can be found in [M03]. Our Generator matrix is construc
that its first column is filled with '1's, this reduces Galois field multi
to simple XOR calculus. Consequently, (i) the first parity bucket of eac
is XOR-encoded, and (ii) in case of one data bucket failure and the fir
bucket is alive, the bucket is recovered using XOR-decoding.

4.1 File Creation

Along a synchronous inserts, where the client waits for a reply before iss
next insert query, the time to create an LH*$_{RS}$ file of 25000 records is 7
for $k = 0$, 9.990 sec for $k = 1$ and 10.963 sec for $k = 2$. We get better perf
results, with the new flow control and acknowledgement strategy desc
§3.2. Indeed, for *Window Size* fixed to 5, the time to create an LH*$_{RS}$ file
records is 4.484 sec for $k = 0$, 6.969 sec for $k = 1$ and 8.109 sec for $k = 2$

4.2 Search Performances

The key search time is the basic referential of access performance of th
type, since it does not involve the parity calculus for $k > 0$. We have m
the time to perform random *individual* (synchronous) and *bulk* (asynch
successful key searches. For synchronized searches, the client waits fo
before issuing another search query. While in asynchronous searches, t
sends a flow of search queries to four data buckets. All measures we
client side. We have measured the search times in a file of 125000 reco
tributed over four buckets. The average individual and bulk search tim
0.2419 ms and 0.0563 ms respectively. Thus the former is about 40 tim
than a disk key search. The latter reaches the speed-up of almost 200 tim
former was bound basically by the server processing speed, while the latt
client speed.

4.3 Data Record Recovery

The record recovery manager is located at one of the parity buckets. First
for the data record key inside the parity bucket structure, sends search q
alive buckets, then waits until receipt of replies to compute the missing
finally it sends the recovered record to the client. We have measured the
times in a file of 125000 records, distributed over four buckets. The t
measured at the parity bucket and starts when the bucket gets the messa
the coordinator, until the recovery of the record. The average data record
time is 1.30 ms using XOR decoding and 1.32 using RS decoding. No
the average time to scan our parity bucket to locate the key c of the da
was measured to be 0.822 ms. This is the dominant part of the total ti
represents 62% and 64% respectively. If one seeks for faster record rec
buckets are much larger, the additional already mentioned index (c, r)
bucket at the parity bucket should help. Notice finally that even the basi
recovery times remain significantly faster than for a disk file.

4.4 High Availability

To measure the recovery performance, we create an LH*$_{RS}$ group wit
buckets, the group contained 125 000 = 4 * 31 250 data records. Table 1
parity bucket (PB) creation times.

	$Total\ Time(sec)$	$Process.\ Time(sec)$	$Com.\ Time$
$1DB-XOR$	2,062	1,484	0
$1DB-RS$	2,103	1,531	0

Table 1: Parity Bucket Creation Times in Seconds.

Notice that, (i) the time to create the first parity bucket (PB-XOR
XORing only, is faster than for the other buckets (PB-RS), using the RS
and (ii) the communication time represents almost 15% of the total time,
[M00][ML02] it represents 60% of the total time.

4.5 Bucket Recovery

To measure the recovery performance, we create an LH*$_{RS}$ group wit
buckets and 1, 2, or 3 parity buckets. The group contained 125 000 = 4
data records. The Slice parameter varies in the set {1250, 3125, 6250
31250}, being respectively {4%, 10%, 20%, 50%, 100%} of a bucket's s
recovery of a single data bucket (DB), can use the first parity bucket an

quently the XOR decoding only. The 1st line of Table 2 presents this cas
natively, the recovery can use another parity bucket, applying the RS de
The 2nd line of the table shows the measures of this case. Our numbers p
efficiency of the LH*$_{RS}$ bucket recovery mechanism. It takes only 1.555
to recover 9.375 MB of data in three buckets.

	$Total\ Time(sec)$	$Process.\ Time(sec)$	$Com.\ Time$
$1DB-XOR$	$0,720$	$0,265$	0
$1DB-RS$	$0,855$	$0,380$	0
$2\ DBs$	$1,162$	$0,600$	0
$3\ DBs$	$1,555$	$0,911$	0

Table 2: Parity Bucket Creation Times in Seconds.

During experiments, in the set of Slice parameter values {1250, 312
15625, 31250}, the recovery performances are almost equal. Table 2 re
average times, for experiments details refer to [M03].

5 Conclusion

We have evaluated an LH*$_{RS}$ file creation time, parity bucket creation,
trieval in both normal mode and degraded mode, and finally the recovery
than one data bucket, with respect to the new bucket architecture. Than
TCP/IP connections handler component, communication times are imp
80% [M03], and no more dominates total times for TCP/IP based scena
periments prove the efficiency of the proposed scenarios to LH*$_{RS}$ sch
validate our devised architecture. Our architectural proposals and scen
independent of the erasure resilient code used and data distribution sche
ther work concerns implementation of other encoding and decoding tech

References

[B00] F. Bennour, Un Gestionnaire de Structures Distribues et Scalab
les multiordinateurs Windows: Fragmentation par Hachage, *PhD
French, Paris Dauphine University*, 2000.

[BB94] M. Blaum, J. Brady, J. Bruck, J. Menon, EVENODD: An
Scheme for Tolerating Double Disk Failures in RAID Architecture
1994.

[BK95] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, D.Zuc
An XOR-Based Erasure-Resilient Coding Scheme, *ICSI Tech. Rep*
048, 1995.

[D01] A W. Diéne, Contribution la Gestion de Structures Distribues et S
PhD thesis in French, Paris Dauphine University, 2002.

[H94] L. Hellerstein, G.A. Gibson, R.M. Karp, R.H. Katz, D.A. Patterso
ing Techniques for handling Failures in Large Disk Arrays, *Algo*
12,pp.182-208, 1994.

[ISI81] Information Sciences Institute, RFC 793: Transmission Control
(TCP) - Specification, Sept. 1981, http://www.faqs.org/rfcs/rfc793.

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

[LS00] W. Litwin, J.E. Schwarz, LH*$_{RS}$, A High-Availability Scalable
uted Data Structure using Reed Solomon Codes, *Proceedings of t*
SIGMOD, p.237-248, 2000.

[M00] R. Moussa, Implantation partielle & Mesures de performa
LH*$_{RS}$, *Universit Paris Dauphine, MSc Report in French*, Octob
http://ceria.dauphine.fr/Rim/dea.pdf.

[M03] R. Moussa, Experimental Performance Analysis of the new
Scenarios and Architecture Design, *CERIA Research Report*, Ju
http://ceria.dauphine.fr/Rim/comparison0603.pdf

[MB00] D. MacDonal, W. Barkley, MS Windows 2000 TCP/IP Implem
Details, http://secinf.net/info/nt/2000ip/tcpipimp.html.

[ML02] R. Moussa, W. Litwin, Experimental Performance Manage
LH*$_{RS}$ Parity Management, *WDAS proceedings pp. 87-98*,2002.

[P97] J. S. Plank, A Tutorial on Reed-Solomon Coding for fault-Tole
RAID-like Systems, *Software - Practise & Experience, 27(9), pp 99*
Sept. 1997.

[PGK88] D. A. Patterson, G. Gibson, R. H. Katz, A Case for Redundar
of Inexpensive Disks, *Proc. of ACM SIGMOD Conf, pp.109-106*, Ju

[R89] M. O. Rabin, Efficient Dispersal of Information for Security, Load
ing and Fault Tolerance, *Journal of ACM, Vol. 26, N 2, pp. 335-3*
1989.

[S02] T. J.E. Schwarz S.J., Reed Solomon Codes for Erasure Corre
SDDS, *WDAS proceedings, pp. 75-86*, 2002.

[SDDS] http://ceria.dauphine.fr/SDDS-bibliographie.html

[WGBC00] M. Welch, S. D. Gribble, E. A. Brewer, D. Culler, A Desig
work for Highly Concurrent Systems, *UC Berkeley Tech. Report U
00-1108*, April 2000.

[WK02] H. Weatherspoon, J. D. Kubiatowicz, Erasure Coding vs. Rep
A quantitative Comparison, *Proceedings of the 1st International W
on Peer-to-Peer Systems, p.328-338*, March 2002.

[XB99] L. Xu, J. Bruck, Highly Available Distributed Storage Systems,
ings of workshop on Distributed High Performance Computing,
notes in Control and Information Sciences, Springer Verlag, 1999.

Rim Moussa , Dr. in Computer Science, Dauphine University, Pl. du Mal de La
75016, France. E-mail: Rim.Moussa@dauphine.fr

Thomas Schwarz , Professor, Santa Clara University, adrress?, USA.
TSchwarz@calprov.org

Acknowledgements This work was partly supported by the research gra
Microsoft Research, and by ? as well as by the scholarship from Tunisian
ment.