Université Paris Dauphine
D.E.A. 127

Mémoire de D.E.A.

# Implantation Partielle & Mesures de Performances de LH*$_{RS}$ : Une SDDS à Haute Disponibilité

Rédigé par :
Rim Moussa

Encadré par :
Pr. Witold Litwin

# Remerciements

Mes remerciements s'adressent au Pr. Witold Litwin pour ses conseils, ses encouragements et sa disponibilité constante.

Je tiens a remercier également M. Jim Gray, T. Schwartz et M. Darell Lang, pour l'intérêt qu'ils portent au travail de l'équipe du CERIA, et les suggestions fructueuses qu'ils m'ont fait part.

Je remercie vivement Fethi, Yakham et Aly pour la sollicitude dont ils m'ont entourée.

Enfin, je remercie mes parents pour leur soutien et surtout pour la patience et la compréhension, dont ils ont fait preuve a mon égard, tout au long de l'élaboration de ce mémoire.

# Table des Matières

# Introduction

Les nécessités de la vie moderne placée, sous le signe de la machine et de la vitesse, ont poussé les chercheurs à transiger avec les systèmes centralisés, et par conséquent, distribuer les données sur différents sites.

Les Structures de données distribuées se sont développées, grâce au progrès réalisé d'une part dans l'infrastructure réseau, et d'autre part l'apparition d'une nouvelle architecture matérielle : les muliordinateurs[1]. Ces derniers sont caractérisés par:

1. Une architecture modulaire, permettant au système d'évoluer indéfiniment en ajoutant des potentiels de calcul et des capacités de stockage,
2. Un meilleur rapport coût / performance.

Cette architecture matérielle exige de nouvelles structures de données, et c'est dans ce contexte que les Structures de Données Distribuées et Scalables[2] [SDDS] ont été proposées.

Malgré les différents avantages, que procure la distribution, la vulnérabilité aux pannes est un problème, qui s'accentue avec l'augmentation du nombre de machines dans le réseau.

La Structure de Données Distribuée et Scalable LH*$_{RS}$ adresse le problème de la tolérance aux pannes, particulièrement l'échec de nœuds, par l'ajout de données redondantes. Ces dernières sont calculées par le code Reed Solomon.

Ce mémoire porte sur la proposition et l'évaluation de scénarios d'augmentation de la disponibilité d'un fichier LH*$_{RS}$, et la récupération des données source et des données redondantes.

Le plan du mémoire s'établit comme suit,

Le chapitre I se consacre à l'état de l'art. Tout au long, de ce chapitre, nous évoquons des travaux de recherche en matière de structures de données distribuées et étudions le mécanisme par lequel elles adressent la problématique de la fiabilité.

[1] Un multiordinateur est une collection d'ordinateurs, stations de travail, connectés par un réseau à haut débit.
[2] Terme ang., signifiant le maintien de performances face à l'augmentation en charge [DeWitt & Gray, 1992].

Le chapitre II, présente les fondements théoriques de la structure de données distribuée à haute disponibilité scalable LH*$_{RS}$. Cette structure peut surmonter l'échec de plusieurs nœuds et utilise les codes de Reed Solomon dans le calcul des données de parité.

Vient ensuite, le chapitre III, où nous proposons des scénarios relatifs à la création du fichier LH*$_{RS}$, l'augmentation de la disponibilité, et un scénario de récupération général.

Enfin, le chapitre IV traite des expérimentations menées afin d'évaluer les performances des scénarios proposés.

# 1 Fiabilité dans les Systèmes de Stockage Distribués

## 1.1 Introduction

Pour assurer la fiabilité 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, à titre non exhaustif, certaines solutions, notamment, en premier plan la technologie RAID, et en second plan, un intérêt particulier est manifesté à l'égard des structures de données distribuées et scalables.

## 1.2 Technologie RAID

La non fiabilité des disques force les gestionnaires des systèmes à faire des copies de données le plus fréquemment possible, et ce pour prévenir contre une éventuelle panne. Pour relever le défi de fiabilité, il y a eu recours aux extra-disques qui contiennent de l'information redondante et servant à une récupération en cas de panne de disque. L'acronyme RAID : *Redundant Arrays of Independent Disks* est apparu pour la première fois en 1988, et a été introduit par D.A Patterson, G. Gibson & R.H.Katz.

L'article original [Patterson et al., 1988] propose une taxonomie de cinq différentes organisations, que sont :

### 1.2.1 RAID niveau 1

C'est l'option la plus coûteuse, 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 (voir figure 1.1). Chaque mise à jour est faite sur les deux disques et quand le disque principal échoue, les utilisateurs basculent automatiquement au miroir.

En mode normal, deux accès en écriture sont requis pour une mise à jour et des lectures en parallèle peuvent être faites. Tandis qu'en mode de reconstruction, un accès en lecture au bon disque et une écriture sur un nouveau disque sont requis pour le remplacement du disque en échec.

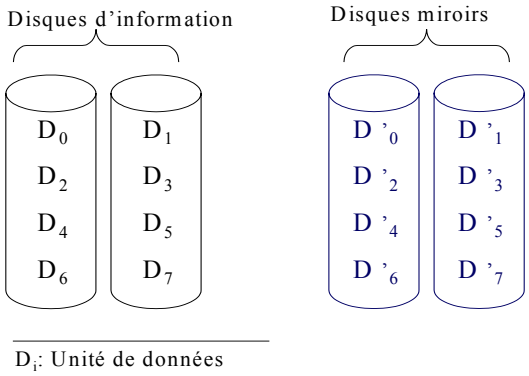


Figure 1.1: Architecture RAID niveau 1

### 1.2.2 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 l'information sur le disque en échec. L'inconvénient de cette organisation, mis à part la question de performance, est le nombre de disques de parité.

### 1.2.3 RAID niveau 3

Ce niveau ne comporte qu'un seul disque de parité. L'inconvénient se situe au niveau de la performance des transactions d'E/S. En effet, la reconstitution d'une unité de transfert nécessite plusieurs E/Ss à différents disques d'un même groupe.

### 1.2.4 RAID niveau 4

Ce niveau remet en cause le niveau 3 et plus particulièrement le fait qu'un transfert soit distribué entre un groupe de disques. Ainsi chaque unité est sauvée sur un seul disque et nous notons alors une nette amélioration en performance des transactions d'E/Ss. Néanmoins, cette organisation présente un risque d'un goulot d'étranglement des écritures sur le disque de parité.

### 1.2.5 RAID niveau 5

En distribuant les unités de parité sur les disques, ce niveau diminue, mais n'élimine pas le goulot d'étranglement des écritures sur les unités de parité. Cette architecture est la plus répondue dans l'industrie de stockage (voir figure 1.2).

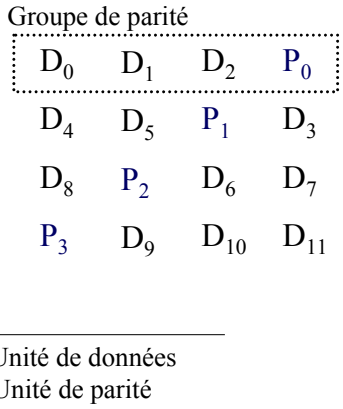


Figure 1.2: Architecture RAID niveau5

### 1.2.6 Combinaison RAID niveau 1 et RAID niveau 5

Mogi et al., [Mogi et al.,1996], soulignent les limites du niveau RAID5, et elles sont autour des deux faits suivants :

1. Le fait qu'une mise à jour d'un bloc requiert quatre accès disque : trois lectures et une écriture. En effet,
   $P_{nouveau} = P_{ancien}$ XOR $D_{ancien}$ XOR $D_{nouveau}$

2. Et le fait que le processus de reconstruction nécessite l'accès à tous les disques du même groupe que le disque en échec. En effet, quand un disque du groupe j échoue, les données perdues sont récupérées par le calcul suivant :

   $D_{j,k} = P_j$ XOR $D_{j,1}$ XOR… XOR $D_{j,k-1}$ XOR $D_{j,k+1}$ XOR … XOR $D_{j,n}$

Etant donné, que le RAID niveau 1 présente de meilleures performances, les auteurs proposent de combiner les niveaux 1 et 5. Et ce, en divisant l'espace de stockage en deux parties : une partie miroir contenant les données correspondant à une importante fréquence d'accès, et une autre partie RAID5 contenant le reste des données. Un processus de migration des données est requis puisque le taux d'accès aux données évolue.

### 1.2.7 EVENODD

[Blaum et al., 1994], proposent une extension du RAID niveau 4, et ce par l'ajout de seulement deux disques de parité à m disques d'information. Ils supposent qu'un disque comporte *m*-1 symboles, ce qui leur permet de placer l'ensemble des symboles de données et de parité dans un tableau à 2-dimensions $(m-1) * (m+2)$.

Les opérations de codage se basent sur le ou-exclusif, et le processus de codage est le suivant :

Le premier disque de parité est juste la parité impaire horizontale,

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

Ceci dit, $a_{lm}$ est le l^iéme symbole du disque $m$, est la parité impaire des l^émes symboles de chaque disque d'information.

Le deuxième disque de parité, est la parité impaire diagonale de pente –1, dont les symboles se calculant comme suit,

$$\begin{cases} \begin{cases} \forall\, l \in [0, m-2], \quad a_{l\,m} = S \quad XOR \quad \left( XOR_{t=0}^{m-1} \quad a_{<l-t>_m\, t} \right) \\ S = XOR_{t=1}^{m-1} \quad a_{m-1-t\, t} \end{cases} \\ < x >_y \leftrightarrow x \bmod y \end{cases}$$

Ci-dessous, un exemple pour $m$ égal à 5,

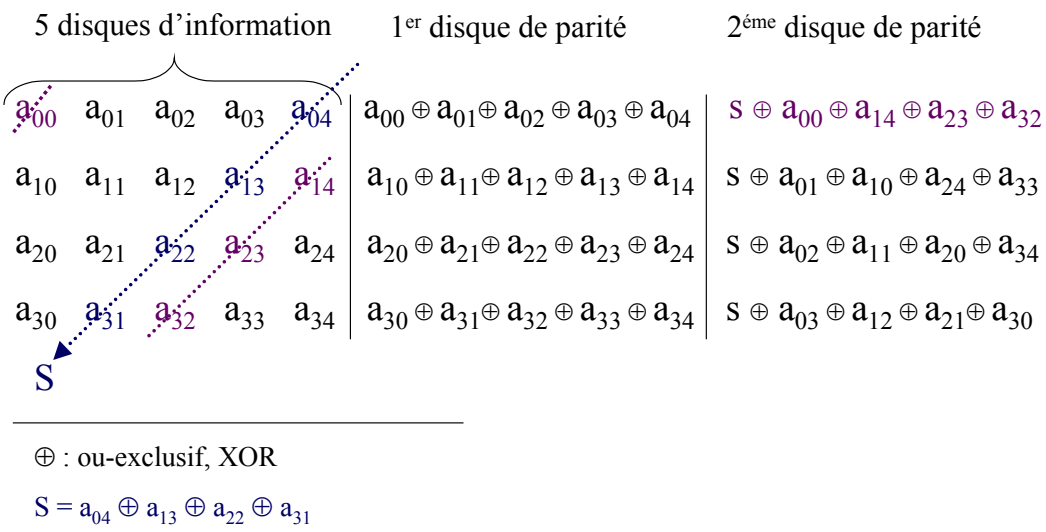


Figure 1.3: EVENODD (m = 5)

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 d'information, se répercute sur les deux symboles de parité de même ligne. 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 X-code a été proposé.

### 1.2.8 X-Code

X-code est une classe de code [Xu et al., 1999], survivant à deux échecs, et dont la 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 d'information, est exactement égale à 2.

Dans X-code, les symboles d'information sont placés dans un tableau *(m-2)*m* et les symboles de parité sont sur les deux dernières lignes. La taille du code est donc *m*m*. Les règles de codage, géométriquement parlant, sont suivant les diagonales de pente 1 et –1.

Les équations calculant les lignes m-1 et m sont les suivantes,

$$\left|\begin{array}{l} \forall\, i \in [0, m-1], \\ \quad \begin{cases} a_{m-2\ i} = XOR_{t=0}^{m-3}\ a_{t\ <i+t+2>_m} \\ a_{m-1\ i} = XOR_{t=0}^{m-3}\ a_{t\ <i-t-2>_m} \end{cases} \\ <x>_y \leftrightarrow x \bmod y \end{array}\right.$$

Ci-dessous une illustration, du X-code pour *m* égal à 5,

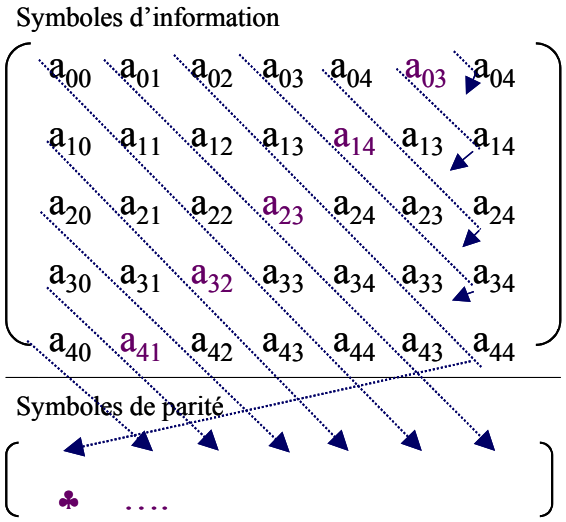


Figure 1.4: X-Code ( *m* = 5)

### 1.2.9 Conclusion

La technologie RAID a fait l'objet de nombreuses autres extensions, par l'intégration de codes linéaires binaires [Hellerstein et al., 1994], et des codes Reed Solomon [Plank, 1997]. Néanmoins, toute expansion du système nécessite une reconfiguration complexe, et s'oppose au critère scalabilité. Ce critère est satisfait par les structures de données distribuées et scalables [SDDS].

## 1.2 Les Structures de Données Distribuées et Scalables

Une Structure de Données Distribuée et Scalable satisfait les trois propriétés clef 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 la suite une dégradation des performances d'accès.

2. La deuxième propriété fondamentale de toute SDDS est que les insertions étendent le fichier sur plusieurs sites, d'une manière incrémentale et transparente pour l'application.

3. Enfin, chaque site est capable de détecter une erreur d'adressage, et de rediriger la requête vers un autre site.

Plusieurs SDDSs ont été proposées. Elles différent par leurs stratégies de distribution de données. La première famille est basée sur la distribution par hachage: LH*. Une deuxième: RP* implante la distribution par intervalles : *Range Partitioning*. D'autres structures sont discutées dans [SDDS], et recensées dans la figure 1.5.

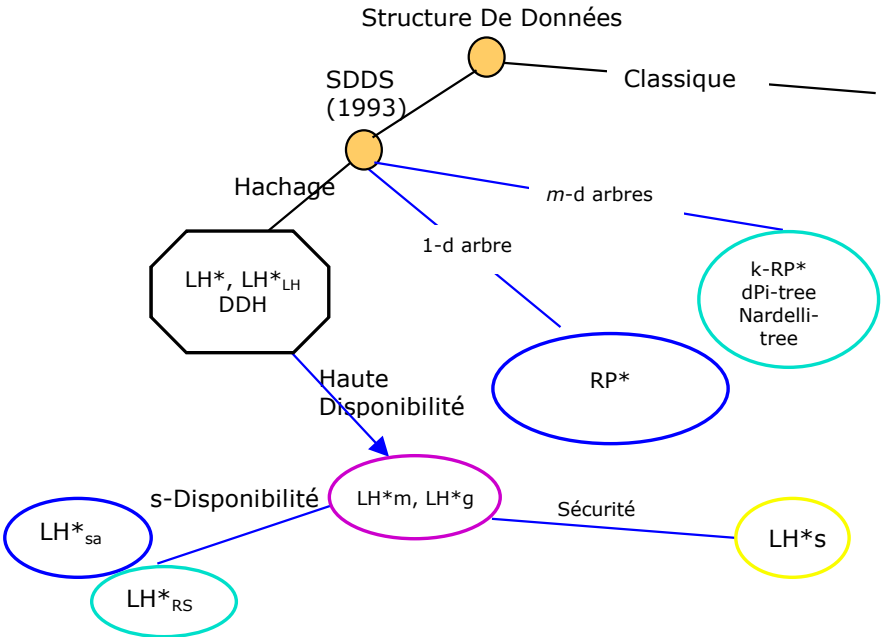


Figure 1.5: Aperçu de différentes SDDSs

### 1.2.1 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.

Un enregistrement est alloué à une case, dont l'adresse logique est déterminée par le calcul de la valeur de la fonction de hachage $h_i(C) = C \bmod N^* 2^i$, tels que C désigne la clé de l'enregistrement, N le nombre de cases initiales, et i le niveau du fichier.

L'algorithme d'adressage logique d'un enregistrement de clé C est le suivant :

$a \leftarrow h_i(C)$

Si a < n alors a ← $h_{i+1}$(C)

Une variable n, dénote un pointeur sur la prochaine case à éclater. Les éclatements se font dans un ordre déterministe : 0 .. N-1, 0 .. 2N-1, … etc., et suite à des débordements de cases, admettant une capacité maximale de b enregistrements.

Un éclatement fait déplacer presque la moitié des enregistrements de la case n à la nouvelle case n + N*$2^i$, et le couple (i,n) caractérisant le fichier LH change comme suit,

n ← n +1
Si n ≥ $2^i$ alors n←0 et i ← i +1.

Sans contrôle d'éclatements, le taux de remplissage du fichier est de 65-69%, il peut être amélioré, en ajoutant des conditions à la condition de surcharge d'une case, 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, par la suite, une fusion de cases, qu'est le phénomène inverse de l'éclatement. Dans ce cas, le couple (i, n) évolue :

n ← n -1
Si n < 0 alors i ← i -1 et n ← $2^i$ - 1.

Dans ce qui suit, nous étudions la fiabilité dans certaines structures de données distribuées et scalables implantant la distribution de données moyennent une fonction de hachage.

### 1.2.2 LH*

Cette SDDS implique trois acteurs, que sont :

- Serveurs
- Clients
- [ Coordinateur ]

Cette structure de données distribuée et scalable, due à Litwin & al en 1993 [Litwin et al., 1993], a fait l'objet de plusieurs travaux de recherche, notamment [Devine, 1993] [Vingralek et al., 1994].

Un fichier LH* : F est sauvé sur plusieurs serveurs (ou nœuds), et à chaque serveur est affectée une seule case de F. Quand une case déborde, elle notifie le coordinateur, qui 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.

Le hachage linéaire suppose qu'un client utilise le couple (i, n) pour le calcul de l'adresse logique d'un enregistrement de clé c, cette hypothèse fait de (i, n) une ressource partagée pour les clients, la sauver au niveau de chaque client et l'actualiser saturera le réseau de messages. LH* pallie le problème, de manière qu'un client ait sa propre image (i', n') de l'état du fichier, et suivant laquelle, il calcule l'adresse *a'*.

L'algorithme exécuté par le client est le suivant,

a' ← $h_{i'}$(C)

Si a' < n' alors $a \leftarrow h_{i'+1}(C)$

La valeur de *a'* risque d'être différente de l'adresse logique correcte a. Ainsi, un serveur recevant un enregistrement de clé C, commence par vérifier si l'adresse de la case relative au fichier F coïncide avec *a*, et si le test échoue, le serveur renvoie l'enregistrement à un autre serveur. Au maximum, un enregistrement de clé C arrive à la case *a*, au bout de deux renvois.

L'algorithme de test & renvoi exécuté par le serveur est le suivant:

$a' \leftarrow h_j(C)$
Si $a' \neq a$ alors $a'' \leftarrow h_{j-1}(C)$
Si $a'' \in ]a, a'[$ alors $a' \leftarrow a''$
Renvoyer C à *a'*

Suite à une erreur d'adressage, le client reçoit une notification : IAM (Image Adjustement Message), lui permettant d'actualiser son image. Le message comporte j et a, étant respectivement le niveau de la case et l'adresse logique de la case correcte. Notons que l'image du client ne devient pas forcément identique à (i, n).

L'algorithme d'ajustage exécuté par le client est :

$i' \leftarrow j - 1$
$n' \leftarrow a + 1$
Si $n' \geq 2^{i'}$ alors $n' \leftarrow 0$ et $i' \leftarrow i'+1$

Une table T stockée au niveau de chaque serveur, et mise à jour par le coordinateur, permet de faire la correspondance entre les adresses logiques des cases et celles physiques des serveurs.

Notons que 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 ont été proposées pour répondre le critère de fiabilité. Ces Structures, recensées dans la figure 1.5, adaptent différents concepts à LH*, que sont,

- Concept de miroir : $LH^*_M$
- Concept de fragmentation: $LH^*_S$
- Concept de groupage : LH*g, $LH^*_{SA}$ & $LH^*_{RS}$.

### 1.2.3 $LH^*_M$

Soient :
p: la probabilité qu'une case soit disponible = 0.99.
$p_c$ : la probabilité qu'un nœud soit disponible = p.

La probabilité qu'un fichier F soit disponible est $p_F = p^N$.

Quand N croit → $p_F$ décroît et converge vers 0. En effet,
$N = 100 \rightarrow p_F = 0.37$,
$N = 1000 \rightarrow p_F = 0.00004$.

Pour pallier cette néfaste convergence, un concept populaire consistant à prévoir un fichier miroir a fait l'objet de l'article [Litwin & Neimat, 1996]. L'existence d'un miroir implique que pour un fichier LH*, un enregistrement existe au moins dans deux cases LH*.

Supposons qu'un enregistrement soit sauvé dans deux cases, appartenant à deux différents serveurs, la probabilité d'un double échec de cases est égale à $(1 – p)^2$. Ainsi $p_c$ est égale à $1 – (1 – p)^2 = 0.99$. Nous notons que quand N croit, $p_F$ ne converge pas aussi vite vers 0. En effet, pour N = 1000, $p_F = 0.91$.

A chaque miroir correspond un ensemble de clients primaires. Et 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 au 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 interrogés par les clients. Dans ce qui suit, nous présentons une variante plus économique de point de vue stockage.

### 1.2.4 $LH^*_S$

Un fichier $LH^*_S$ est constitué de k+1 fichiers segments LH*, étant $S_1 \ldots S_{k+1}$. Le fichier segment de parité $S_{k+1}$ permet la récupération de données perdues. Pour insérer un enregistrement R, de clé C et d'attributs la séquence de bits: $B = b_{1..} b_k b_{k+1} b_{2k} \ldots b_{mk}$, le client procède comme suit :

- Dans une première étape, il crée k fragments, k > 1 tel que le fragment i = si est la concaténation des bits $b_i b_{k+i} b_{2k+i} \ldots$, auxquels est ajoutée la clé C, le fragment si est envoyé par la suite au segment Si, considéré comme un fichier LH*.

- 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 $b_j' = XOR_{j \in [1, k]} b_{ij}$, $b_{ij}$ désigne le bit appartenant au fichier segment Si et étant à la jéme position, $s_{k+1}$ est envoyé alors au fichier segment $S_{k+1}$.

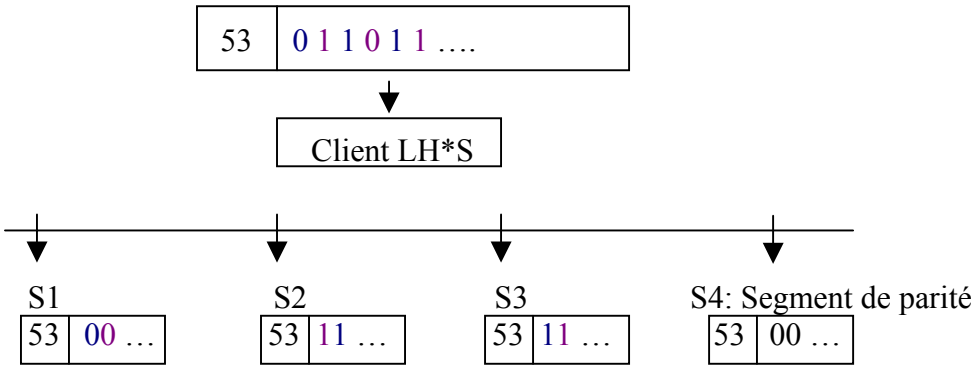


Figure 1.9: Insertion d'un enregistrement dans LH*s (k = 3)

$LH^*_S$ supporte n'importe quelle indisponibilité d'une case, il est 1-disponible. Il présente également un aspect de sécurité renforcé. En effet, chaque case n'a qu'un bit de k bits

consécutifs de B. Désignons par *S* le nombre de bits manquants pour un intrus, ce dernier a $2^S$ possibilités pour reconstituer un fragment, *s* étant égale à | B | (1-1/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'une case LH*, et ce sous certaines conditions.

### 1.2.5 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 [Litwin & Rish, 1997], [Lindberg, 1997]
- LH*$_{SA}$ [Litwin & al., 1998]
- LH*$_{RS}$ [Litwin & Schwarz, 1999], [Ljongstrom, 2000]

Une comparaison entre ces trois SDDSs est faite dans l'article [Litwin & al., 1999].

- LH*g

Un fichier LH*g est une paire de deux fichiers LH*, F1 : Fichier primaire et F2 : Fichier de parité.
Une case m $\in$ F1, appartient au groupe de cases g = Int (m/k), tel que k étant le nombre maximal de cases dans un groupe de cases.

m $\in$ [0 ... k –1] $\rightarrow$ groupe g = 0
k $\in$ [k … 2k –1] $\rightarrow$ groupe g = 1
:

Au niveau de chaque case *m* de F1, est définie une variable r, dite compteur d'insertion.

Structure d'un enregistrement de données

| C | **g** | attributs non-clé |
|---|---|---|

Structure d'un enregistrement de parité

| **g** | $C_1 \ldots C_l$ tq $l \leq k$ | Bits de parité |
|---|---|---|

**g** = (g, r) tel que, g = Int (m/k) et r est le rang.

Le champs Bits de parité d'un enregistrement de parité est calculé tel qu'il est la concaténation de l bits pi, pi = XOR bi, bi étant le i$^{éme}$ bit de champs attributs non clé correspondant à $C_{i:1..l}$.

Suite à l'éclatement d'une case, un enregistrement de données a une probabilité de 50% de migrer à la nouvelle case créée. Même après sa migration, le champs g de l'enregistrement ne

change pas, donc l'enregistrement reste logiquement rattaché à un groupe, ainsi les enregistrements de parité ne sont pas mis à jour.

Le contenu de F2 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.

- LH*$_{SA}$

La haute disponibilité de LH*$_{SA}$ se matérialise par le fait qu'un enregistrement de données appartient à plusieurs groupes de parité, ces groupes sont formés de telle manière qu'ils ne partagent qu'une seule case.

## 1.3 Conclusion

Tout au long de ce chapitre, nous avons exposé différentes structures de données distribuées, et étudié le mécanisme par lequel, elles assurent la fiabilité. Le chapitre suivant présente les fondements théoriques de la SDDS à haute disponibilité scalable LH*$_{RS}$.

# 2 LH*$_{RS}$: Fondements Théoriques

## 2.1 Introduction

Le présent chapitre, détaille les fondements théoriques de la structure de données distribuée à haute disponibilité scalable : LH*$_{RS}$, [Litwin & Schwartz, 2000]. Cette structure survit à l'échec de plusieurs nœuds, par l'introduction de données de parité. Le calcul de parité dans LH*$_{RS}$ se base sur les codes Reed Solomon ( [Lin et al, 1983], [Blomer et al., 1995], [Rizzo, 1996], [Rizzo, 1997]). Dans une première section, nous exposons la structure d'un fichier LH*$_{RS}$, puis nous montrons la haute disponibilité scalable, pour détailler par la suite le calcul de parité.

## 2.2 Structure d'un fichier LH*$_{RS}$

### 2.2.1 Groupage

LH*RS est une structure de données distribuée et scalable à haute disponibilité, implantant la concept de groupage. Un fichier LH*$_{RS}$ est subdivisé en groupes, tel qu'un groupe est un ensemble de *m* cases de données. La haute disponibilité dans LH*$_{RS}$ se matérialise par l'ajout de *k* cases de parité à un groupe.

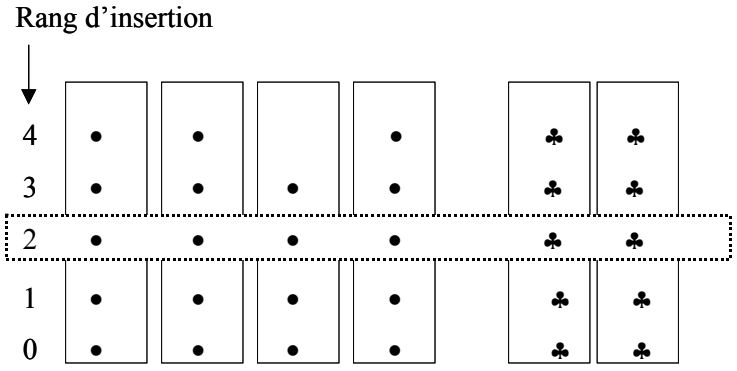


Figure 2.1: Un groupe d'un fichier LH*$_{RS}$ (m = 4, k = 2)

Le figure 2.1 illustre un groupe de quatre cases de données, et auquel sont rattachées deux cases 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 contient tous les enregistrements de données ayant le même rang dans le même groupe. Les enregistrements de parité de clé *r* sont calculés à partir d'un groupe d'enregistrements de données de même rang et même groupe.

### 2.2.2 Structure des Enregistrements

Dans ce qui suit, nous détaillons la structure d'un fichier LH*RS,

Un enregistrement de données a la structure suivante,

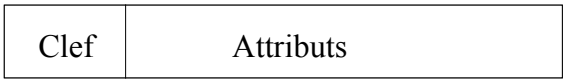


Figure 2.2: Structure d'un Enregistrement de Données

Un enregistrement de parité a la structure suivante,

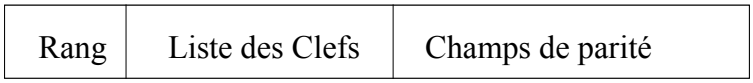


Figure 2.3: Structure d'un Enregistrement de Parité

tel que,

- **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 pour rang d'insertion *Rang*, dans les cases auxquelles ils appartiennent. La section suivante porte sur le calcul de parité.

## 2.3 Haute disponibilité scalable

Un fichier est crée avec une case de données et une case de parité (figure 2.4 (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.

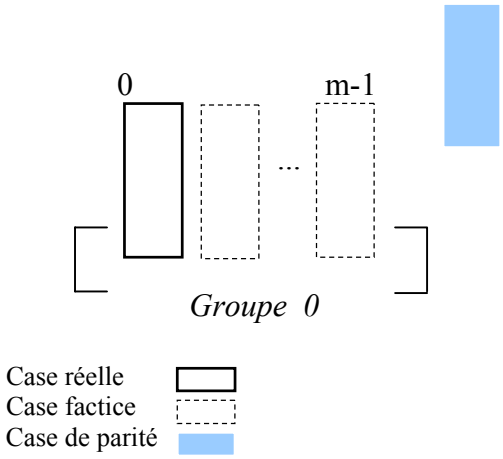


Figure 2.4 (a) : Fichier LH*RS sur une case de données

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 sur deux cases de données (figure 2.4 (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 insertions.

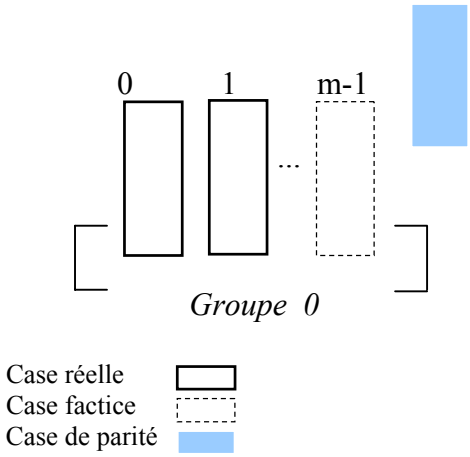


Figure 2.4 (b) : Fichier LH*RS sur deux Cases de Données

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 ajoutent des cases de données nouvelles 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 2.4 (c)).

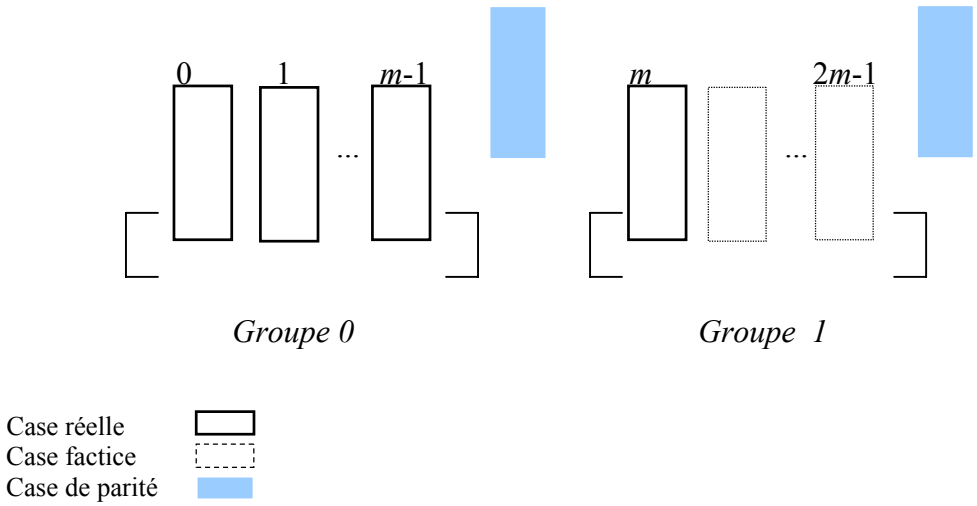


Figure 2.4 (c) : Fichier LH*RS sur *m+1* Cases de Données

Quand la taille du fichier atteint M = $2^{i1}$, 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 0.

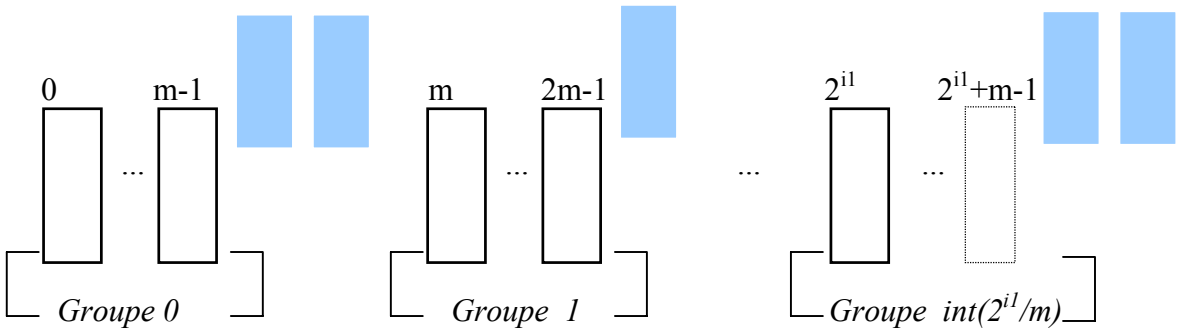


Figure 2.4 (d) : Fichier LH*RS (k = 2)

Quand le fichier double de taille $2^{i1}$*2, tous les groupes auront deux cases de parité, ainsi le fichier devient 2-disponible et peut 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. Les valeurs de i1, i2 etc. qui déterminent quand le niveau de disponibilité doit augmenter sont contrôlées par le coordinateur.

[Litwin & Schwartz, 2000], proposent deux approches permettant de définir M, au-delà duquel, le fichier étant k-disponible devient 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 $m^k$ cases. Dans la notation ci-dessus, nous aurons quand M est égal à,

  $2^{i1} = m^1$ → fichier 1-disponible,
  $2^{i2} = m^2$ → fichier 2-disponible,
  ...
  $2^k = m^k$ → fichier k-disponible etc.

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

- La seconde stratégie de contrôle de fiabilité détermine *i* de manière dynamique. Ainsi, quand 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, une $k+1^{éme}$ case de parité est créée.

# 2.4 Calcul de Parité

## 2.4.1 Les codes Reed Solomon

La problématique est la suivante, sachant *m* enregistrements de données, nous voulons définir un algorithme de codage de ces *m* enregistrements de données en *n* enregistrements, et tel que la récupération des *m* enregistrements de données est possible sachant *m* enregistrements.

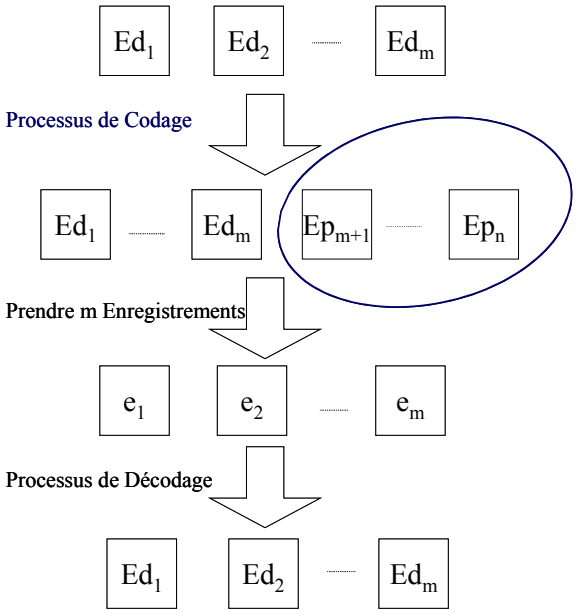


Figure 2.5: Illustration des processus de codage et décodage

La figure 2.5 montre un processus de codage, qui partant de *m* enregistrements de données, calcule *n* – *m* enregistrements de parité. Le processus de décodage se matérialise par la

reconstruction des *m* enregistrements de données, et ce sachant au moins *m* enregistrements de parité ou de données. Les codes de Reed Solomon répondent à la problématique.

Dans ce qui suit, nous montrons le processus de codage te de décodage au niveau symbole.

- Processus de codage

Les codes Reed Solomon sont des codes linéaires. En effet, le code peut être représenté par le système linéaire suivant :

$$\underline{u} = G * \underline{a}$$

$$\Leftrightarrow$$

$$\begin{cases} [u_0 \quad u_1 \quad \cdots \quad u_i \quad \cdots \quad u_n] = [a_0 \quad a_1 \quad \cdots \quad a_{m-1}] * \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{m-1,0} & g_{m-1,1} & \cdots & g_{m-1,n-1} \end{bmatrix} \\ u_i = \sum_{j=0}^{n-1} aj * g_{i,j} \end{cases}$$

$\underline{a}$ : désigne le vecteur de symboles de données,
*G(m*n)* : Matrice génératrice,
$\underline{u}$ : désigne le vecteur des symboles résultats.

Notons que, le code est dit systématique si *G* est la concaténation de la matrice *Im(m*m*), étant la matrice d'identité, et une matrice *P(m*(n-m)*).

- Processus de décodage

Le processus de décodage se fait par la résolution du système linéaire suivant,

$$\underline{b} = H * \underline{a}$$

*tel que*,

$$\begin{cases} b = (b_0, ..., b_{m-1}), \quad bi \in \underline{u} \\ H(m * m) \end{cases}$$

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

$\underline{b}$ : désigne un vecteur de *m* symboles disponibles,
*H* : Sous Matrice de G,
$\underline{a}$ : désigne le vecteur de symboles de données.

### 2.4.2 Calcul des Enregistrements de parité

Nous supposons que les enregistrements de données ont la même taille, autrement nous ajoutons des bits nuls. Nous traitons le champs *Attributs*, (Figure 2.2), d'un enregistrement de données comme une chaîne de bits, que nous découpons en symboles. Un enregistrement de données devient la concaténation de *t* symboles.

Les enregistrements de parité pour le groupe d'enregistrements sont générés à partir des enregistrements de données, un symbole à la fois.

Etant donné, *m* enregistrements de données, nous calculons, comme suit le $i^{éme}$ symbole dans tous les enregistrements de parité simultanément. Comme indique la figure 2.5, à chaque itération, nous procédons comme suit,

- Dans une première étape, nous formons le vecteur *a* ($s_{1j}$, $s_{2j}$, ..., $s_{ij}$, … , $s_{mj}$) où $s_{ij}$ désigne le $j^{éme}$ symbole du $i^{éme}$ enregistrement,

$$
\begin{array}{lcccccccc}
e_1 & = & s_{11} & s_{12} & \dots & \boxed{s_{1j}} & \dots & s_{1t} \\
e_2 & = & s_{21} & s_{22} & \dots & s_{2j} & \dots & s_{2t} \\
 & & & \vdots & & & & \\
e_i & = & s_{i1} & s_{i2} & \dots & s_{ij} & \dots & s_{it} \\
 & & & \vdots & & & & \\
e_m & = & s_{m1} & s_{m2} & \dots & s_{mj} & \dots & s_{mt} \\
 & & & & & \uparrow & & \\
 & & & & & a & &
\end{array}
$$

Figure 2.6 (a) : Etape1 du processus de codage

- Dans une deuxième étape, nous multiplions le vecteur *a* par la matrice génératrice *G[I|P]*, pour obtenir à chaque itération le vecteur *u*

$$u = a * G$$

$$
\begin{array}{lcccccccc}
u & = & \boxed{u_{11}} & u_{21} & u_{31} & \dots & u_{m1} & \boxed{u_{m+11}} & \dots & u_{n1} \\
u & = & u_{12} & u_{22} & u_{32} & \dots & u_{m2} & u_{m+12} & \dots & u_{n2} \\
 & & & & & \vdots & & & & \\
u & = & u_{1i} & u_{2i} & u_{3i} & \dots & u_{mi} & u_{m+1i} & \dots & u_{ni} \\
 & & & & & \vdots & & & & \\
u & = & u_{1t} & u_{2t} & u_{3t} & \dots & u_{mt} & u_{m+1t} & \dots & s_{nt} \\
 & & \downarrow & & & & & \downarrow & & \\
 & & \text{Enregistrement} & & & & & \text{Enregistrement} & & \\
 & & \text{de données 1} & & & & & \text{de parité 1} & &
\end{array}
$$

Figure 2.6(a) : Etape2 du processus de codage

Notons que, chaque symbole de parité du i$^{éme}$ enregistrement de parité est le résultat de la multiplication du vecteur *a* par la i$^{éme}$ colonne de P. Tout l'enregistrement de parité correspond à l'itération de cette multiplication par tous les symboles de données.

Notons que les symboles sont codifiés sur un champs de galois. En effet, en arrondissant ou tronquant les résultats des opérations de multiplications et de division, nous risquons de perdre de l'information, et par conséquent fausser le processus de décodage. C'est ainsi, que l'utilisation des champs de galois trouve sa justification. En effet, un champ est fermé sous l'addition et la multiplication, ce qui signifie que le résultat de l'addition ou de la somme de deux éléments du champs, est un élément du champs.

### 2.4.3 Champs de Galois

Nous utilisons les champs de Galois GF($2^f$), tel que f > 1. Les éléments du champs peuvent être considérés comme des polynômes de degré f-1. L'addition et la soustraction de deux éléments se réduisent au ou-exclusif : XOR.

La multiplication est définie en se basant sur la représentation des éléments de GF($2^f$) en polynômes de degré f, et de coefficients sur le champs GF(2) = {0, 1}. La multiplication est alors la multiplication des polynômes, et le résultat est réduit modulo un polynôme générateur p($\alpha$) irréductible, de degré f.

Une propriété intéressante d'un champs de galois, est qu'il existe au moins un élément particulier du champs, noté $\alpha$, dont les puissances génèrent tous les éléments non nuls du champs.

Notons que pour faciliter les opérations de multiplication et de division, des tables sont utilisées.

### 2.4.4 La Matrice Génératrice G

La matrice *G* est la concaténation de deux matrices *Im* et *P*, tel que *Im (m*m)* est la matrice d'identité. Elle est générée par une transformation élémentaire des lignes de la matrice de Vandermonde *V(m*n).*

La matrice de Vandermonde se présente comme suit,

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

En appliquant, l'algorithme suivant nous obtenons G.

```
Pour  toute colonne i : 0 ... m-1  Faire
   Si   m[i, i] = 0  Alors
        chercher j > i tel que m[j, i] ≠ 0
        mj ↔ mi
   Fin si
```

mi ← m[i, i]$^{-1}$ mi
Pour toute ligne j : 0 ... m-1, tq j ≠ i Faire
mj ← mj - m[j, i] mi

D'autres matrices peuvent être utilisées, telle que la matrice de Cauchy [Rabin,1989] [Blomer et al., 1995].

## 2.5 Exemple de calcul de parité

### 2.5.2 GF(16)

Pour les opérations de multiplication et division, nous utilisons la table suivante,

| Chaîne de bits | Entier | Hexadécimal | log |
|---|---|---|---|
| 0000 | 0 | 0 | - ∞ |
| 0001 | 1 | 1 | 0 |
| 0010 | 2 | 2 | 1 |
| 0011 | 3 | 3 | 4 |
| 0100 | 4 | 4 | 2 |
| 0101 | 5 | 5 | 8 |
| 0110 | 6 | 6 | 5 |
| 0111 | 7 | 7 | 10 |
| 1000 | 8 | 8 | 3 |
| 1001 | 9 | 9 | 14 |
| 1010 | 10 | A | 9 |
| 1011 | 11 | B | 7 |
| 1100 | 12 | C | 6 |
| 1101 | 13 | D | 13 |
| 1110 | 14 | E | 11 |
| 1111 | 15 | F | 12 |

Les opérations de multiplication et de division deviennent,

$$g * h = \text{antilog}_{\alpha} ( (\log_{\alpha} (g) + \log_{\alpha} (h)) \bmod 2^4 - 1 )$$

$$g / h = \text{antilog}_{\alpha} ( (\log_{\alpha} (g) - \log_{\alpha} (h)) \bmod 2^4 - 1 )$$

### 2.5.2 Calcul des Enregistrements de Parité

Dans ce qui suit, nous fixons *m* à 4 et nous utilisons GF(16).

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 8 & F & 7 & 1 & 7 & 9 & 3 & C & 2 & A & E & 7 & 7 \\ 0 & 1 & 0 & 0 & F & 8 & 1 & 7 & 9 & 7 & C & 3 & A & 2 & 7 & E & 7 \\ 0 & 0 & 1 & 0 & 1 & 7 & F & 8 & 3 & C & 7 & 9 & E & 7 & 2 & A & 7 \\ 0 & 0 & 0 & 1 & 7 & 1 & 8 & F & C & 3 & 9 & 7 & 7 & E & A & 2 & 7 \end{bmatrix}$$

Supposons que nous avons les quatre enregistrements de données suivants,

*ed1* « En arche … »
*ed2* « Dans le … »
*ed3* « Am anfrang … »
*ed4* « In the beginning … »

Leurs correspondants un codage ASCII, en notation hexadécimale, sont :

*ed1* « 4 5 6 E 2 0 4 1 7 2 … »
*ed2* « 4 1 6 D 2 0 4 1 6 E … »
*ed3* « 4 4 6 1 6 E 7 3 2 0 ... »
*ed4* « 4 9 6 D 2 0 4 1 6 E … »

↓

*a* = (4, 4, 4, 4) ...

* G

*u* = (4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 0)
*u* = (5, 1, 4, 9, F, 8, A, 4, 3, 1, 1, 2, 7, E, 9, 9, A)
: : ...

↓ ↓

ed1 ep1

Figure 2.7: Exemple de Codage m = 4, GF(16)

Remarquons que, nous n'avons pas besoin de calculer toutes les coordonnées de chaque *u* du même coup. En effet, nous pouvons reconstituer un enregistrement à la fois. La $i^{éme}$ coordonnée de *u* est égale à *a* * *Gi*, tel que *Gi* désigne la $i^{éme}$ colonne de *G*.

# 2.6 Récupération des Enregistrements

## 2.6.1 Algorithme de récupération

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 suit,

1. Collecter n'importe quels m enregistrements disponibles du segment,

2. Concaténer les colonnes correspondantes dans *G*, pour obtenir une matrice *H (m*m)*,
3. Inverser *H*, $H \xrightarrow{Pivot\ de\ Gauss} H^{-1}$
4. Restituer les symboles des m enregistrements,
5. Itérer sur *i* : 1 .. *t*,
   5.1 Faire déplacer une variable vecteur b, dont les éléments sont les $i^{émes}$ symboles des *m* enregistrements,
   5.2 Multiplier b par $H^{-1}$ pour obtenir a.

## 2.6.2 Exemple de récupération

Dans ce qui suit, nous fixons *m* à 4, nous utilisons GF(16), et nous supposons que seuls le quatrième enregistrement de données et les trois premiers enregistrements de parité soient disponibles. L'algorithme de récupération procède comme suit,

- Constituer *H* en concaténant les colonnes $G_4$, $G_5$, $G_6$ *et* $G_7$, et l'inverser,

$$G = \left[\begin{array}{cccc|cccc|cccccccc} 1 & 0 & 0 & 0 & 0 & 8 & F & 7 & 1 & 7 & 9 & 3 & C & 2 & A & E & 7 & 7 \\ 0 & 1 & 0 & 0 & F & 8 & 1 & 7 & 9 & 7 & C & 3 & A & 2 & 7 & E & 7 \\ 0 & 0 & 1 & 0 & 1 & 7 & F & 8 & 3 & C & 7 & 9 & E & 7 & 2 & A & 7 \\ 0 & 0 & 0 & 1 & 7 & 1 & 8 & F & C & 3 & 9 & 7 & 7 & E & A & 2 & 7 \end{array}\right]$$

$$H = \begin{bmatrix} 0 & 8 & F & 1 \\ 0 & F & 8 & 7 \\ 0 & 1 & 7 & 8 \\ 1 & 7 & 1 & F \end{bmatrix} \xrightarrow{\text{Pivot de Gauss}} H^{-1} = \begin{bmatrix} B & F & A & 1 \\ C & 4 & 2 & 0 \\ 4 & 7 & D & 0 \\ 2 & D & 4 & 0 \end{bmatrix}$$

- Restituer les symboles des *m* enregistrements

```
        i : 1  …        t
ed4   « 4  9   … »
ep1   « 4  F   … »
ep2   « 4  8   … »
ep3   « 4  A   … »
        ↓
        b = (4, 4, 4, 4)      ...
    * H-1  ____________________

        a = (4, 4, 4, 4)
        a = (5, 1, 4, 9)
        :    :
             ↓    ↓
           ed1 .. ed4
```

# 2.7 Scénario réel de manipulations-client versus calcul parité

La procédé de calcul des enregistrements de parité décrit dans la section (§2.4.2), n'est pas réel. En effet, les cases de parité sont générées suite aux manipulations client. Les manipulations-client élémentaires sur les enregistrements de données sont: l'insertion, la suppression et la mise à jour.

Concernant, le calcul de parité, 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.

## 2.7.1 Mise à Jour

Considérons une mise à jour du i$^{éme}$ enregistrement de données dans son groupe, soit *a* le vecteur de symboles de même rang des enregistrements de données dans le groupe d'enregistrements avant la mise à jour, et *a'* le même vecteur après la mise à jour. Les vecteurs *a* et *a'* différent seulement à la position *i*.

Soient *u* et *u'*, tels que $u = a * G$ et $u' = a' * G$.

La différence est égale à $\Delta = u - u' = (a - a') G$.

Désignons par $\Delta i$ la différence en terme de symboles de même ordre des deux vecteurs, donc

$$a - a' = (0, \ldots, \Delta i, \ldots, 0)$$

$$u' = a' G = (a + (a - a')) G = a G + (a - a') G = u + \Delta u$$

Pour calculer $\Delta u$, nous avons seulement besoin de la i$^{éme}$ ligne Li. En XORant $\Delta u$ à *u*, nous obtenons *u'*, et par consèquent le contenu de chaque enregistrement.

$$\begin{array}{lcccl}
(a - a')_1 * G & = & \boxed{\Delta i1} * Li & = & \Delta u1 \; (\;\; , \quad \ldots , \quad\quad , \ldots) \\
(a - a')_2 * G & = & \Delta i2 * Li & = & \Delta u2 \; (\;\; , \quad \ldots , \quad\quad , \ldots) \\
\vdots & & \vdots & & \\
(a - a')_t * G & = & \boxed{\Delta it} * Li & = & \Delta ut \; (\;\; , \quad \ldots , \quad\quad , \ldots)
\end{array}$$

$$\begin{array}{ccc}
\downarrow & \downarrow & \downarrow \\
\Delta\text{-enregistrement} & vd1 & vp1 \\
 & \underline{\text{XOR } ed1} & \underline{\text{XOR } ep1} \\
 & ed1_{nouveau} & ep1_{nouveau}
\end{array}$$

La case *i* calcule le Δ-enregistrement, et envoie l'ensemble du rang *r* de l'enregistrement à mettre à jour, numéro *i* de la case et le Δ-enregistrement aux cases de parité. Ces dernières recalculent alors leurs enregistrements de parité de clé *r*.

### 2.7.2 Exemple

La première insertion de l'enregistrement de données *ed1* dans la case de données 0 est une mise à jour d'un enregistrement factice, le Δ-enregistrement est obtenu en XORant des bits nuls à *ed1*

La procédure de génération des enregistrements de parité est illustrée ci-dessous,

*ed1 « En arche ... »* en hexadécimal « 4 5 6 E … »,

L1 (1, 0, 0, 0, 8, F, 1, 7, 9, 3, C, 2, A, E, 7, 7)

$(a-a')_1 * G = \boxed{4} * L1 = \Delta u1\ (4, 0, 0, 0, 6, 9, \ldots)$
$(a-a')_2 * G = \boxed{5} * L1 = \Delta u2\ (5, 0, 0, 0, E, 6, \ldots)$
$\vdots \quad\quad \vdots$
$(a-a')_t * G = \boxed{\Delta it} * L1 = \Delta ut\ (\ , 0, 0, 0, \ , \ , \ldots)$

↓ Δ-enregistrement

↓ vd1 XOR ed1 = $ed1_{nouveau}$

↓ vp1 XOR ep1 = $ep1_{nouveau}$

Nous insérons l'enregistrement « Dans le … », en hexadécimal « 41 6D 20 41 6E … » dans la case de données 1, qui est également vide. Cette chaîne de symboles est envoyée aux deux cases de parité en tant que Δ-enregistrement, la procédure de génération des enregistrements de parité est illustrée ci-dessous.
Nous utilisons L2, puisque l'enregistrement « Dans le … » est le deuxième dans le groupe 0.

L2 (0, 1, 0, 0, F, 8, 7, 9, 7, C, 3, A, 2, 7, E, 7)

$(a-a')_1 * G = \boxed{4} * L2 = \Delta u1\ (0, 4, 0, 0, 9, 6, F, \ldots)$
$(a-a')_2 * G = \boxed{1} * L2 = \Delta u2\ (0, 1, 0, 0, F, 8, 7, \ldots)$
$\vdots \quad\quad \vdots$
$(a-a')_t * G = \boxed{\Delta it} * L2 = \Delta ut\ (0\ , \ , 0, 0, \ , \ , \ldots)$

↓ Δ-enregistrement

↓ vd1 XOR ed1 = $ed1_{nouveau}$

↓ vp1 XOR ep1 = $ep1_{nouveau}$

Le premier enregistrement de parité est mis à jour de la sorte,

« 9 F 4 7 …»
XOR « 6 E 5 9 …»

$ep1_{new}$ « F 1 1 E …»

Supposons qu'après insertion de deux autres enregistrements de données, les champs B correspondant aux enregistrements de parité *ep1* et *ep2* deviennent,

*ep1* « 4 F 6 3 6 E E 4 … »
*ep2* « 4 8 6 E DCE E … »
...
A présent, nous voulons mettre à jour le premier enregistrement de données, il devient « In initio … », nous calculons tout d'abord le Δ-enregistrement, ce dernier est envoyé aux enregistrements de parité.

$$\begin{array}{ll} & ed1_{ancien} \quad \text{« 4 5 6 E 2 0 4 1 ... »} \\ \text{XOR} & ed1_{nouveau} \quad \text{« 4 9 6 E 2 0 6 9 … »} \\ \hline \Delta\text{-enregistrement} & \text{« 0 C 0 0 0 0 2 8… »} \end{array}$$

$$\begin{array}{lcccl} (a-a')_1 * G & = & \boxed{0} * L1 & = & \Delta u1\ (0, 0, 0, 0, 0, 0, \ldots) \\ (a-a')_2 * G & = & C * L1 & = & \Delta u2\ (C, 0, 0, 0, A, 8, \ldots) \\ \vdots & & \vdots & & \\ (a-a')_t * G & = & \Delta it * L1 & = & \Delta ut\ (\ , 0, 0, 0, \ , \ , \ldots) \end{array}$$

$$\begin{array}{ccc} \downarrow & & \\ \Delta\text{-enregistrement} & & \end{array} \qquad \begin{array}{cc} \downarrow & \downarrow \\ vd1 & vp1 \\ \underline{\text{XOR } ed1} & \underline{\text{XOR } ep1} \\ ed1_{nouveau} & ep1_{nouveau} \end{array}$$

## 2.8 Conclusion

A présent, nous connaissons les fondements théoriques de LH*RS. Nous pouvons dès lors aborder le prochain chapitre.

# 3 Proposition de Scénarios

## 3.1 Introduction

Le présent chapitre expose la création d'un fichier LH*RS et les manipulations élémentaires, propose deux scénarios d'augmentation de la disponibilité du fichier par l'ajout de cases de parité, et un scénario de récupération général.

Tout au long du présent chapitre, nous adoptons la terminologie suivante:

*m*: la 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 $l^{éme}$ colonne de *G*,
*t* : taille du champs de parité d'un enregistrement de parité,
%, mod : opérateur modulo,
+, - dans un champs de galois : XOR,
* : multiplication dans un champs de galois,
x + = y ⇔ x ← x + y, idem pour x – = y,
!= ⇔ ≠,

## 3.2 Architecture

Une première version de $LH*_{RS}$ a été proposée dans [Ljungström, 2000]. LH*RS consiste en l'ajout cases de parité à une variante de LH* : $LH*_{LH}$ [Karlson et al., 1996], implantée sous SDDS-2000 [Bennour, 2000] (figure 3.1).

### 3.2.1 SDDS-2000

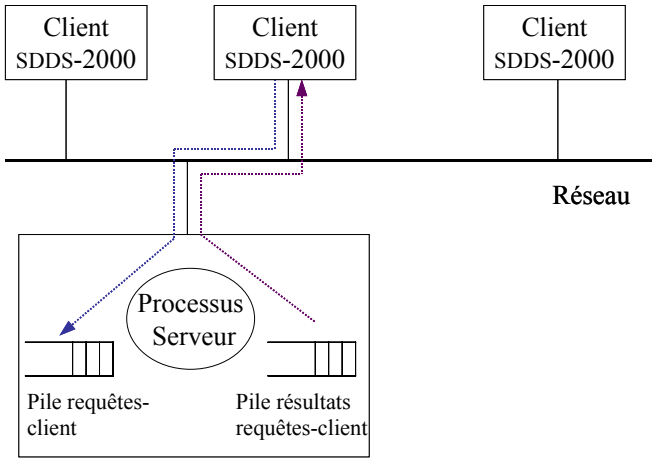


Figure 3.1: Architecture globale de SDDS-2000

- Serveur SDDS

Le serveur de SDDS-2000 fonctionne en mode concurrent, et se base sur la mise en œuvre du multi-tâches sous Windows NT, par l'utilisation des processus légers[3].

Un premier processus léger d'écoute[4] sur un port du réseau, adopte le fonctionnement suivant:

E1 - Attendre l'arrivée d'une requête émise par un client SDDS,
E2 - Mettre la requête dans une file d'attente puis signaler cet événement,
E3 - Retourner à E1.

Un groupe de processus légers de travail[5] adoptent le fonctionnement suivant:

T1 - Attendre l'événement signalant l'arrivée d'une requête,
T2 - Prendre une requête de la file d'attente,
T3 - Analyser la requête pour identifier le traitement correspondant,
T4 - traiter ou rediriger la requête,
T5 - retourner à T1.

- Client SDDS

Un client SDDS-2000 assure les fonctions réception et analyse de la requête. Le client calcule l'adresse du serveur auquel la requête doit être envoyée en se basant sur son image. Une fois l'adresse déterminée, le client envoie la requête au serveur adéquat en utilisant les Win sockets. Notons que, le client reste à l'écoute des accusés de réception de la part des serveurs. La réponse attendue peut être accompagnée d'un message d'ajustement de l'image du client.

[3] Processus léger, traduction de Thread.
[4] Processus léger d'écoute, traduction de listen thread.
[5] Processus léger de travail, traduction de Worker thread.

### 3.2.2 LH*$_{LH}$

Une case de donnée LH*$_{LH}$ utilise deux niveaux d'indexations. Le premier est un niveau d'indexation réseau géré par l'algorithme LH* qui permet au client de trouver le serveur LH* : case de données LH*, contenant l'information désirée. Le deuxième est un niveau d'indexation interne géré par l'algorithme LH. La structure d'une case LH*$_{LH}$ est illustrée dans la figure 3.2.

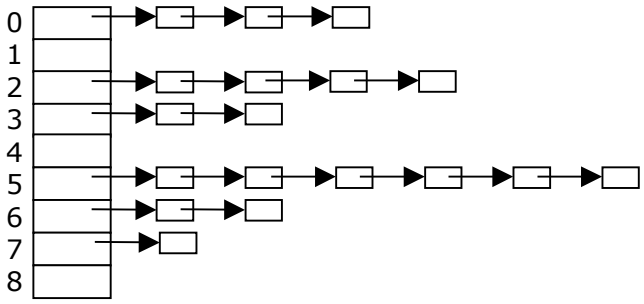


Figure 3.2: Architecture interne d'une case LH*$_{LH}$

Une case de parité a la même structure interne qu'une case de données. Notons que qu 'une case de parité subit l'éclatement interne des cases LH, et ne subit pas l'éclatement réseau.

## 3.3 Scénario de Manipulation Client

Ce scénario a été mis en œuvre par Ljungström [Ljungström, 2000]. La mise à jour est une généralisation de la répercussion de toute manipulation client, (insertion, suppression ou mise à jour d'un enregistrement de données), sur les enregistrements de parité, de clés le rang occupé par l'enregistrement de données en question.

La figure 3.4 suppose que l'image du client est bonne, et que l'enregistrement manipulé est destiné à l'insertion, la suppression ou la mise à jour dans la case de données d. La manipulation client se répercute sur l 'ensemble des cases de données du groupe auquel la case de données d appartient.

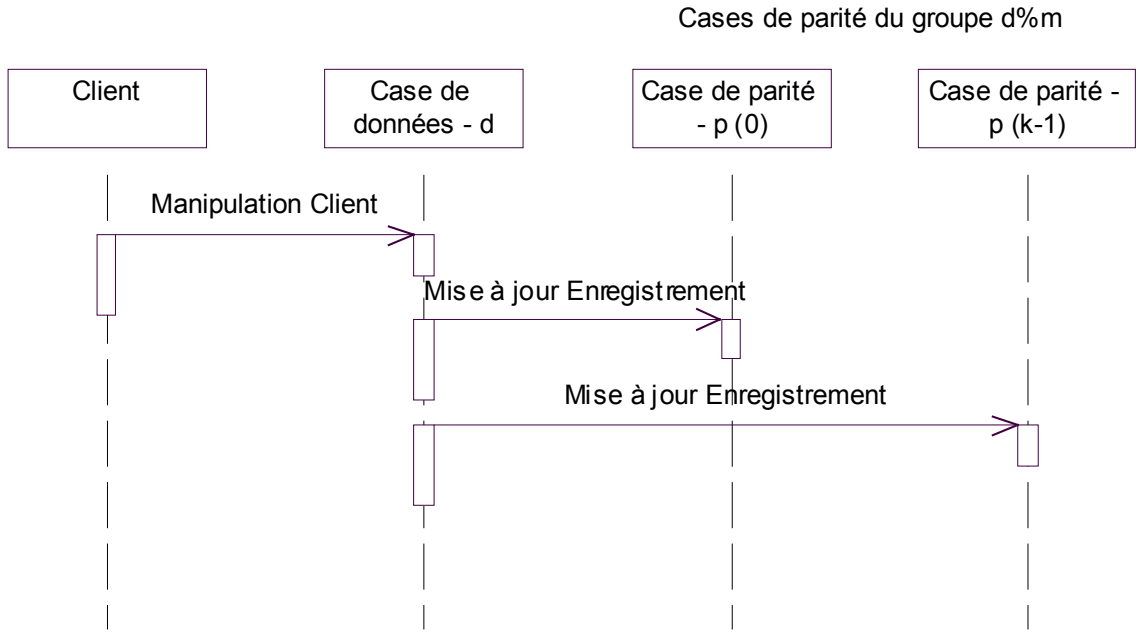


Figure 3.4: Scénario manipulation client

Dans ce qui suit, nous détaillons le traitement spécifique aux manipulations du client, étant, insertion, suppression ou mise à jour d'un enregistrement de données Ed, et son impact sur les cases de parité du groupe.

### 3.3.1 Insertion

Le message d'insertion comporte :

*d* : adresse logique de la case de données émettrice,
*Ed. Clé* : clé de l'enregistrement de données *Ed*,
*Ed. Attributs* : champs attributs de l'enregistrement de données *Ed*,
*rang* : rang occupé par *Ed* dans la case *d*.

A la réception d'un message d'insertion, la case de parité, de niveau l, commence par chercher l'enregistrement de parité *Ep* de clé *rang*. Nous distinguons deux cas,

Cas1: *Ep* n'existe pas
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* * $G_l$,
Incrémenter le nombre d'enregistrements de la case.

Cas2 : *Ep* existe
Il s'agit alors de mettre à jour *Ep*,
*Ep. Liste des clés* [d mod m] ← *Ed. Clé*,
Ep. Parité + = *Ed.Attributs* * $G_l$.

Le calcul du champ de parité de *Ep* s'effectue comme suit, *Ed. Attributs* est une chaîne de symboles de 8 bits, et il s'agit d'itérer sur le nombre de symboles de *Ep.Parité* supposé de taille fixe. A chaque itération, un symbole de *Ep.Parité* est mis à jour comme suit,

pi : $i^{éme}$ symbole de *Ep.Parité*,
di : $i^{éme}$ symbole de *Ed*,
pi += di * G [d mod m] [l].

La mise à jour d'un symbole nécessite, en moyenne, une opération XOR et une opération de multiplication GF. Par conséquent, la mise à jour d'un champs de parité est de t opérations XORs et t opérations de multiplication GF.

### 3.3.2 Suppression

Le message de suppression comporte :
*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 de l'enregistrement de données *Ed*,
*rang* : 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 trouvé, ce dernier est mis à jour, de la sorte:

Ep. Liste des clés [d mod m] ← -1, ça se traduit par Ed devient factice,
Ep. Parité – = Ed. Attributs * G [d mod m] [l].

Outre ce traitement, 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. Il faudrait également prévoir un traitement de « nettoyage » d'une case de parité.

### 3.3.3 Mise à jour

D'abord, la case de parité, de niveau l, recherche l'enregistrement de parité de clé *rang*, étant *Ep*. Puis, met à jour *Ep. Parité*, comme suit,

*Ep. Parité* += *Ed.Attributs'* * G [d mod m][l] – *Ed. Attributs* * G [d mod m][l]

Vu que cette opération, nécessite deux opérations de XOR et deux multiplications GF par symbole, donc 2*t opérations XORs et 2*t opérations de multiplication GF, elle est optimisée de la sorte, dans une première étape, nous calculons ⌊Ed, égal à *Ed. Attributs' – Ed. Attributs,* coûtant t opérations XORs. Ainsi, Ep.Parité = ⌊Ed * G[d mod m][l], coûtant t opérations XORs et t opérations de multiplication GF, pour un total de 2*t opérations XORs et t opérations de multiplication GF. De cette manière nous économisons t opérations de multiplication GF.

Le message de suppression comporte alors,
*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*.

## 3.4 Scénario d'éclatement d'une case de données

Dans un premier temps, nous explicitons l'éclatement dans LH*LH par un scénario, puis nous annotons le scénario d'éclatement LH*RS proposé par Ljungström [Ljungström, 2000], pour enfin proposer un scénario d'éclatement LH*RS plus fiable.

### 3.4.1 Scénario d'éclatement 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. La figure 3.5 illustre le scénario d'éclatement LH*LH.

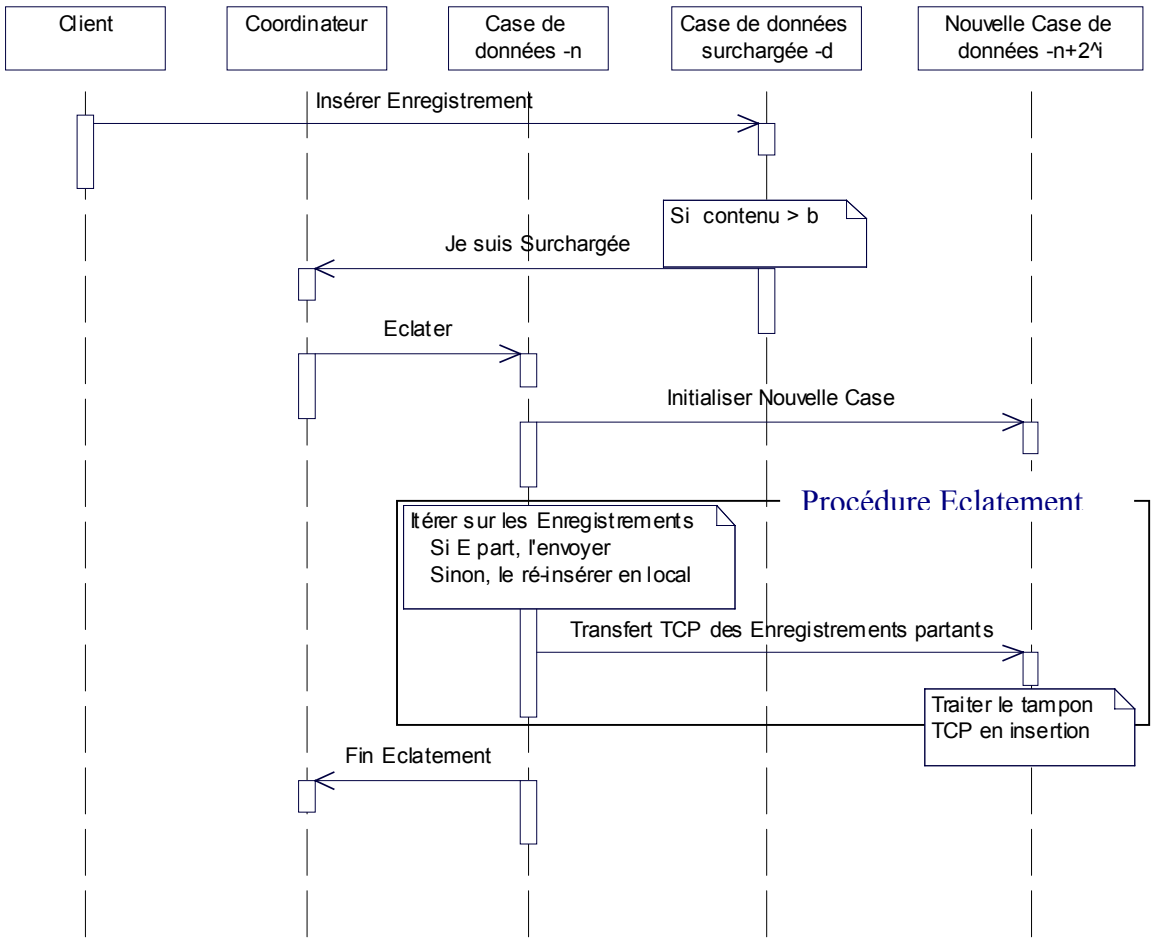


Figure 3.5: Scénario Eclatement LH*LH

Ce qui diffère, dans LH*RS est la procédure d'éclatement, qui est plus complexe. LH*RS implique des cases de parité à mettre à jour.

### 3.4.2 Scénario d'éclatement d'une case de données LH*RS [Ljungström, 2000]

Dans ce qui suit, nous décrivons le scénario d'éclatement LH*RS, puis nous étudions ses différentes facettes.

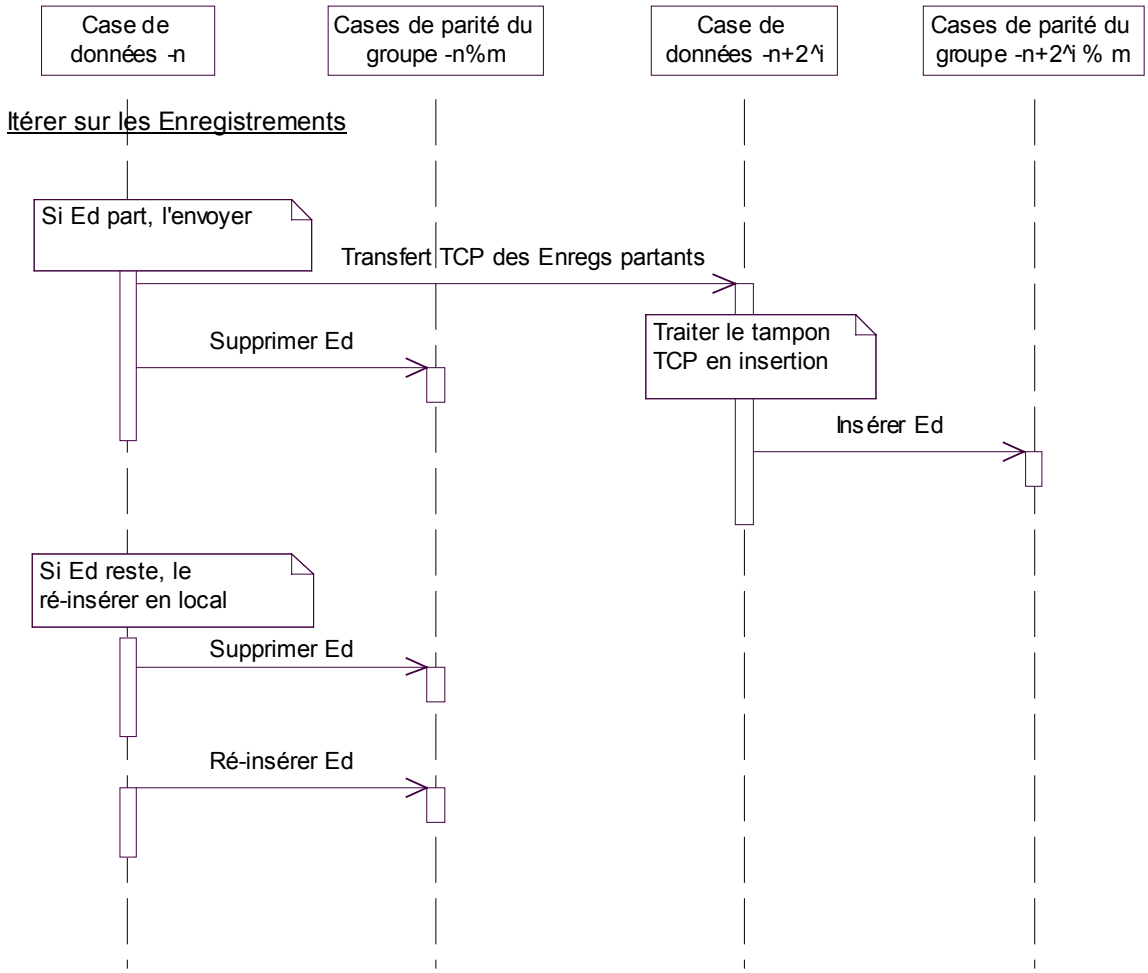


Figure 3.6: Scénario Eclatement LH*RS [Ljungström, 2000]

Au moment de l'éclatement, la case de données LH*, désignée par le coordinateur pour éclater, transfère la moitié de son contenu, approximativement b/2 enregistrements de données, à la nouvelle case.

A l'issue de ce scénario, chacune des cases de parité du groupe, de numéro n mod m, reçoit b messages de suppression et b/2 messages d'insertion, donc un total de 1.5 * b messages. De surcroît, chacune des cases de parité du groupe, de numéro $n+2^i$, reçoit b/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*b messages de mises à jour.

Ce nombre explosif de messages, moyennant le protocole UDP pour être acheminés à leurs destinataires, ne peuvent pas être traités. Certains 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, complexes à repérer et corriger. Ainsi, nous sommes en face de cases de parité inconsistantes qu'il faudrait régénérer.

Si nous admettons que le protocole utilisé pour acheminer ces messages est fiable, à un instant t, il y a une consistance des cases de parité ou une divergence minime par rapport aux contenus des cases de données. Nous entendons par la qualification fiable, assurer transmission du message, et respect de l'ordre entre les messages.

Une nette amélioration réduisant le nombre d'enregistrements consisterait 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 dans le cas où l'enregistrement de données ne migrerait pas vers la nouvelle case. Ce message se présenterait comme suit, supprimer au rang r, et réinsérer au rang r'. Cette alternative réduirait le nombre de messages destinés aux cases de parité du groupe, de numéro n mod m, à b messages, donc le réduit de 1/3. De même, si les deux groupes coïncident, nous pouvons unifier les messages de suppression et d'insertion, puisqu'ils portent sur le même enregistrement, et sont destinés, dans ce cas particulier, aux même cases de parité. Notons que, dans ce cas, le nouveau rang sera calculé à partir d'un compteur sur le nombre d'enregistrements partant.

### 3.4.3 Scénario d'éclatement d'une case de données LH*RS Proposé

Dans ce qui suit, nous illustrons le scénario d'éclatement d'une case de données LH*RS, puis nous expliquons le déroulement du scénario au niveau de chaque entité impliquée.

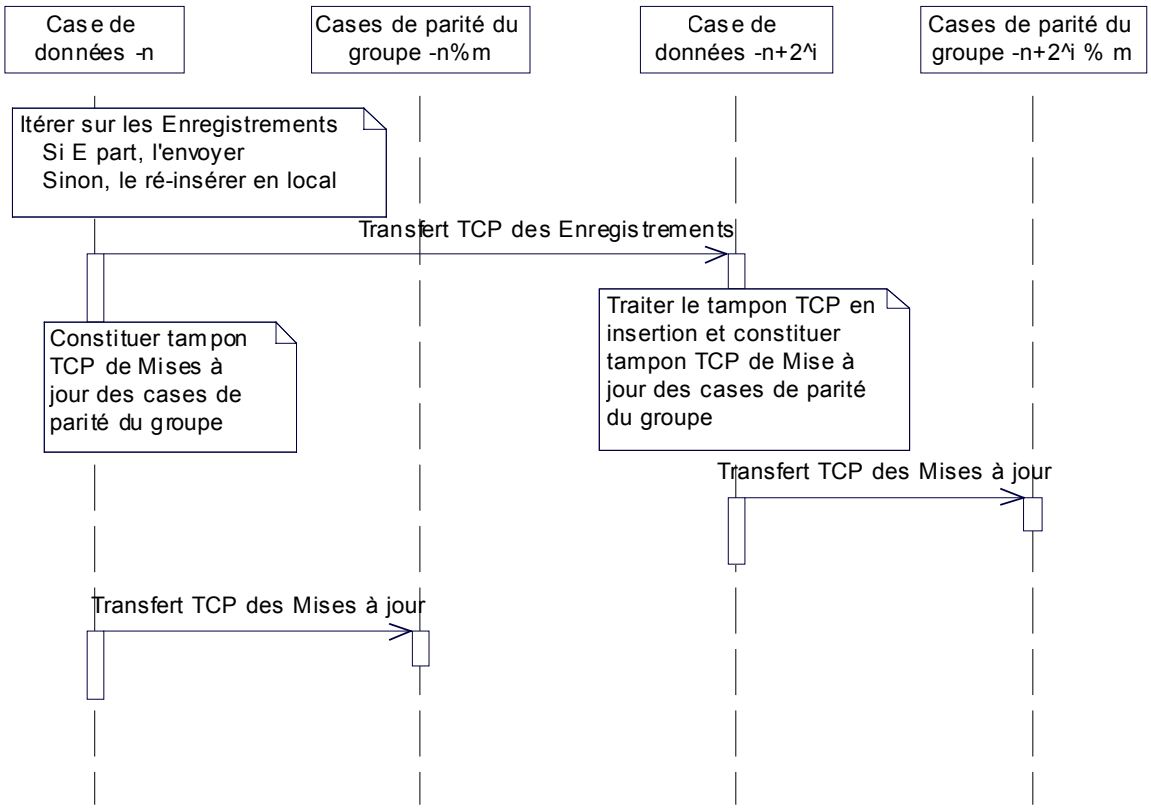


Figure 3.7: Scénario Eclatement LH*RS proposé

**1. Niveau de la case de données *n***

Tout en parcourant la case LH*LH, pour transférer les enregistrements de données partant à 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.*

**2. Niveau case de parité**

Il est à noter qu'une case de parité traite chaque structure *S* du tampon TCP de mise à jour de la sorte,

Si *S. Rang* $\neq -1$
Alors Supprimer de l'enregistrement de parité de clé *Rang,*
l'enregistrement de données *Ed* (§4.2.2)
Fin si

Si *S. Nouveau rang* $\neq -1$
Alors Insérer dans l'enregistrement de parité de clé *Nouveau rang,*
l'enregistrement de données *Ed* (§4.2.1)
Fin si

**3. Niveau 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.

Il est à noter que si la case qui éclate et la nouvelle case appartiennent au même groupe, les deux tampons sont synchronisés. En effet, celui issu de la case qui éclate doit précéder celui issu de la nouvelle case.

**Limites & éventuelles améliorations du scénario proposé**
Le transfert en TCP de la case de données n à la case de données $n+2^i$, se fait enregistrement par enregistrement; si le traitement, se faisait de la même manière, au lieu de l'ensemble des enregistrements transmis en entier, nous noterons certainement de meilleures performances. Idem, pour les mises a jour de parité, ce cas est plus complexe, puisqu'une case de données, que ça soit, celle qui éclate ou la nouvelle, doit pouvoir maintenir plusieurs connexions TCP à la fois ou procéder à un multicast.

## 3.5 Scénario d'ajout d'une case de parité à un groupe

### 3.5.1 Motivation

Nous soulignons l'intérêt de l'ajout d'une case de parité à un groupe de cases de données, vu que quand la taille du fichier augmente, la probabilité que multiples échecs surviennent devient plus importante. Nous justifions que l'ajout d'une case de parité à un groupe de cases de données, minimise cette probabilité d'échec, comme suit,

Nous nous limitons au cas d'un groupe de m cases de données et k cases de parité, ainsi le groupe s'étend sur $m+k$ serveurs. Nous désignons par $p$, la probabilité qu'un serveur est disponible, par conséquent, la probabilité que les m+k serveurs soient disponibles est $p^{m+k}$. Notons que plus $m+k$ augmente, $p^{m+k} << p$. Pour pallier la néfaste convergence de $p^{m+k}$ vers 0, pouvant rendre un groupe irrécupérable, nous ajoutons un serveur de parité à ce groupe,

Dans ce qui suit, nous commençons par présenter le scénario d'augmentation du niveau de disponibilité d'un groupe (figure 3.8), et consistant simplement en l'ajout d'une case de parité au groupe. Puis, nous proposons deux scénarios implantant chacun un protocole de communication différent.

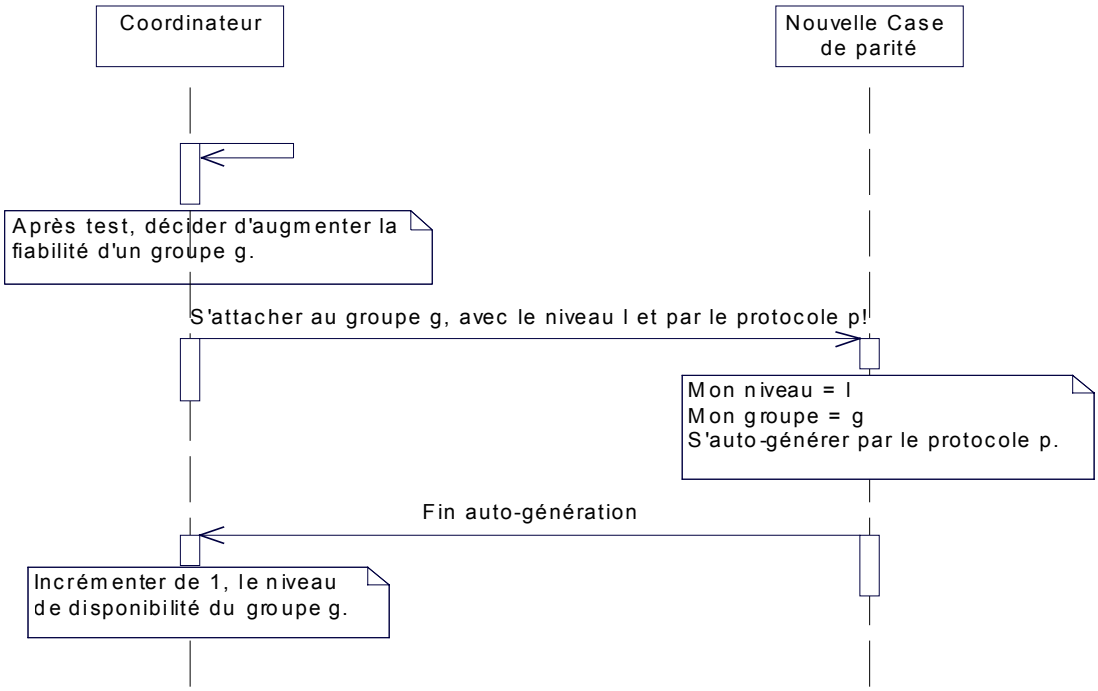


Figure 3.8: Scénario d'ajout d'une case de parité

Il est à signaler, que la procédure test, n'a pas été implantée dans le prototype.

### 3.5.2 Scénario d'ajout d'une case de parité à un groupe par UDP

La nouvelle case de parité « s'auto-génère », 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 (figure 3.9).

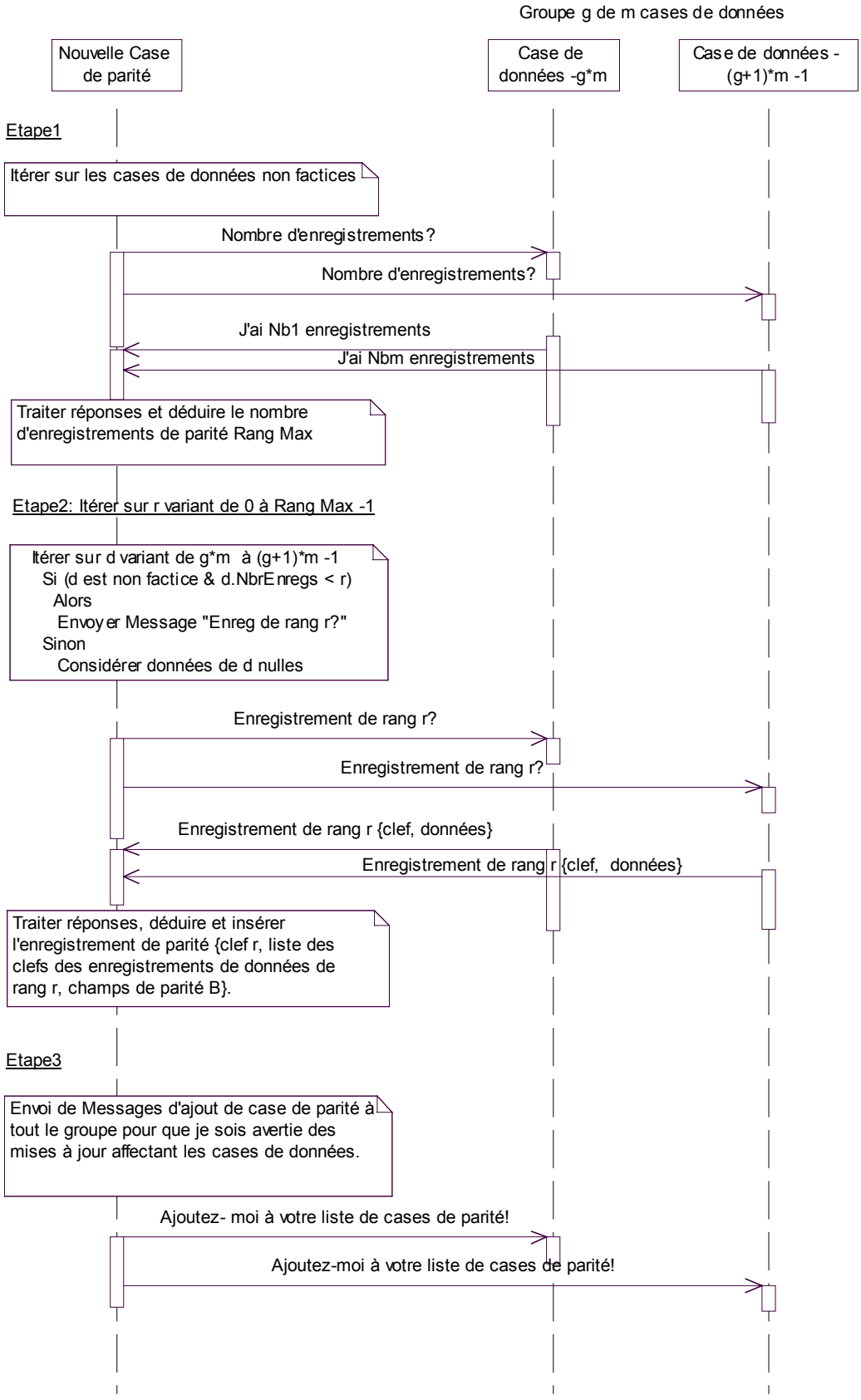


Figure 3.9 : Scénario ajout d'une case de parité par UDP

**Avantages & Inconvénients**

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 la case de parité pour qu'elle demande les récentes manipulations ayant affecté chaque case de données.

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.

### 3.5.3 Scénario d'ajout d'une case de parité à un groupe par TCP

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. La case de parité traite alors l'ensemble des enregistrements en insertion (§4.2.1).

L'interaction entre la nouvelle de case de parité et une case de données d non factice du groupe g se présente comme suit,

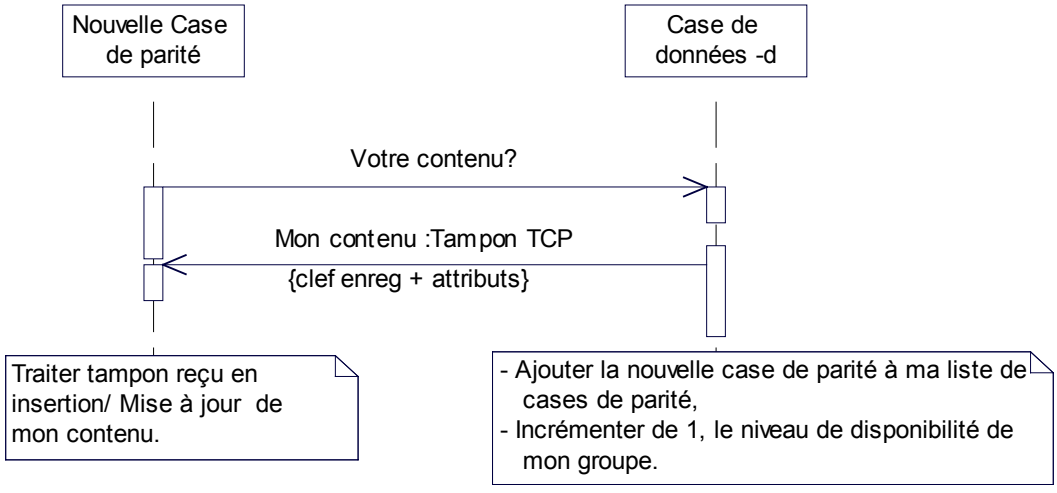


Figure 3.10: Scénario d'ajout d'une case de parité par TCP

**Avantages & Inconvénients**

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 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 ? Les expérimentations montrent des cas à distinguer dépendant du nombre d'enregistrements dans les cases de données.

De même, si le fichier augmente de taille entre temps, et que la case de donnée supposée factice n'est plus désormais le cas, la meilleure solution est que le coordinateur retarde l'éclatement jusqu'à réception du message « Fin auto-génération !», issu de la nouvelle case de parité ou que la nouvelle case de parité n'en tient compte qu'après avoir fini son « auto-génération ».

## 3.6 Scénario de récupération

Dans cette section, nous proposons des scénarios de récupération de cases de données, cases de parité, et récupération mixte. Dans un premier temps, nous discutons le contrôle d'échecs de sites, puis nous explicitons le corps des procédures de récupération, pour enfin montrer la procédure de récupération au niveau de la case gestionnaire de récupération.

### 3.6.1 Contrôleur d'échecs

- Détection d'un échec d'une case de données

Nous distinguons deux cas :

1. Cas où le client veut que le gestionnaire de fichier LH* lui accuse réception de sa manipulation,

   Dans ce cas, quelle que soit la manipulation client, insertion, suppression ou recherche d'un enregistrement, si le délai d'attente maximum est dépassé, le client adresse sa requête au coordinateur. Ce dernier procède au test de la disponibilité de toutes les cases de données et de parité du groupe, soupçonné en échec.

2. Cas où le client ne veut pas que le gestionnaire de fichier LH* lui accuse réception de sa manipulation,

   Dans ce cas, l'échec d'une case de données ne peut pas être perçu par le client suite à une insertion ou suppression. En effet, il ne peut s'apercevoir d'un échec qu'au moment d'une recherche d'un enregistrement; par conséquent, il envoie sa requête de recherche au coordinateur, et ce dernier diagnostique le groupe en question.

- **Détection d'un 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, le coordinateur est averti. Ce dernier, 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.

Vu la complexité de ces scénarios, et les différentes variantes à implanter, et sachant que le prototype a pour but d'évaluer la complexité de la récupération, dans la mesure où c'est possible, de case(s) de données, case(s) de parité ou à la fois cases de données et cases de parité, nous simplifions le scénario de récupération. 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 montrent la procédure de récupération au niveau, du coordinateur et de la case gestionnaire de récupération.

### 3.6.2 Procédure de récupération : niveau coordinateur

Une fois, notifié d'un échec d'un groupe *g*, le coordinateur procède à la vérification de la disponibilité des cases de données et des cases de parité du groupe g; pour enfin, en cas d'échec et possibilité de récupération, désigner un gestionnaire de récupération.

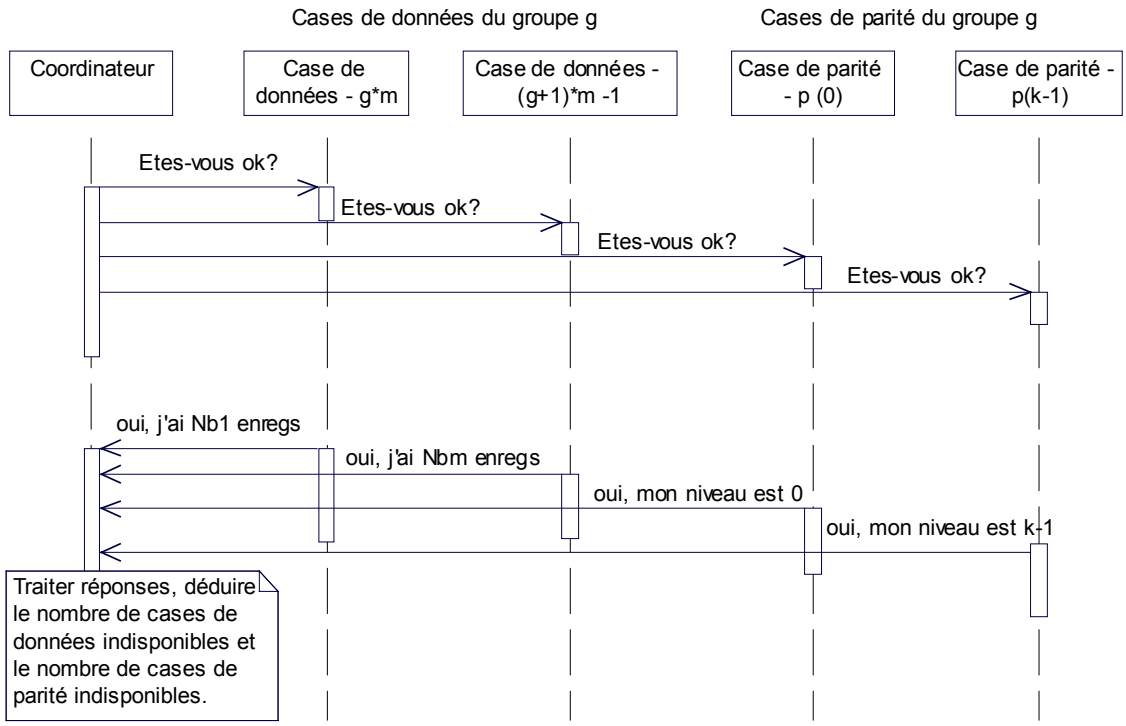


Figure 3.11: Test d'échecs

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, c'est à dire le nombre de cases de parité.

Dans une première étape, le coordinateur teste la possibilité de récupérer. Ainsi, nous distinguons trois cas :

Cas 1: $nd = 0$ et $np = 0$
Il s'agit alors d'une fausse alerte,

Cas 2: $m - nd + k - np < m \Leftrightarrow k - nd - np < 0$
Dans ce cas, le groupe est irrécupérable.

Cas 3: $1 - nd - np \geq 0$, récupération possible.

Suite au troisième cas, 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, le gestionnaire de récupération serait une case de donnée, et case de parité autrement.

Si $np = k$ Alors
Toutes les cases de parité sont indisponibles, le gestionnaire de récupération serait la première case de donnée ayant répondu au test de détection d'échecs, lancé par le coordinateur.

Sinon
Le gestionnaire de récupération serait la première case de parité, ayant répondu au test de détection d'échecs, lancé par le coordinateur.
Fin si

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.

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.

### 3.6.3 Récupération des enregistrements de données

Cette procédure, sachant *m* enregistrements de données ou de parité soient-ils, récupère *nd* enregistrements de données, de même rang, perdus.

*H (m*m)* : fusion de *m* colonnes de *G*, correspondant aux cases disponibles,
Inverser *H*.
*E (m*l)* : structure contenant mes m enregistrements disponibles,
Itérer sur j variant de 0..*t*
$b \leftarrow Ej,$

Pour *i* de 0 à *m*
$$R[i][j] = XOR_{s=0}^{m-1} \; b[s] * H^{-1}[s][i]$$

Fin Itérer

Cette procédure recalcule les m enregistrements de données. Une optimisation de ce calcul serait de ne pas calculer tous les symboles, et remplacer les instructions encadrées par les suivantes,

.

Pour *i* de 0 à *nd*
$$R[i][j] = XOR_{s=0}^{m-1} \; b[s] * H^{-1}[s][Indices[i]]$$
La structure Indices dénote les niveaux des cases de données en échec.

A chaque itération, nous économiserons le calcul de *(m – nd)* symboles, coûtant *(m - nd) * (m-1)* XORs et *(m - nd) * m* multiplications GF.

### 3.6.4 Récupération des enregistrements de parité

Cette procédure, sachant les m enregistrements de données, récupère *np* enregistrements de parité, de même rang, perdus.

*D(m*l)* : 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.

## 3.6.5 Gestionnaire de récupération

### 3.6.5.1 Case de Données

Rappelons que ce cas correspond, à la récupération de l'ensemble des cases de parité du groupe, il est détaillé dans le scénario ci-dessous. Ce dernier montre, d'une part l'interaction du gestionnaire de récupération avec une case de données, et d'autre part l'interaction avec les cases.

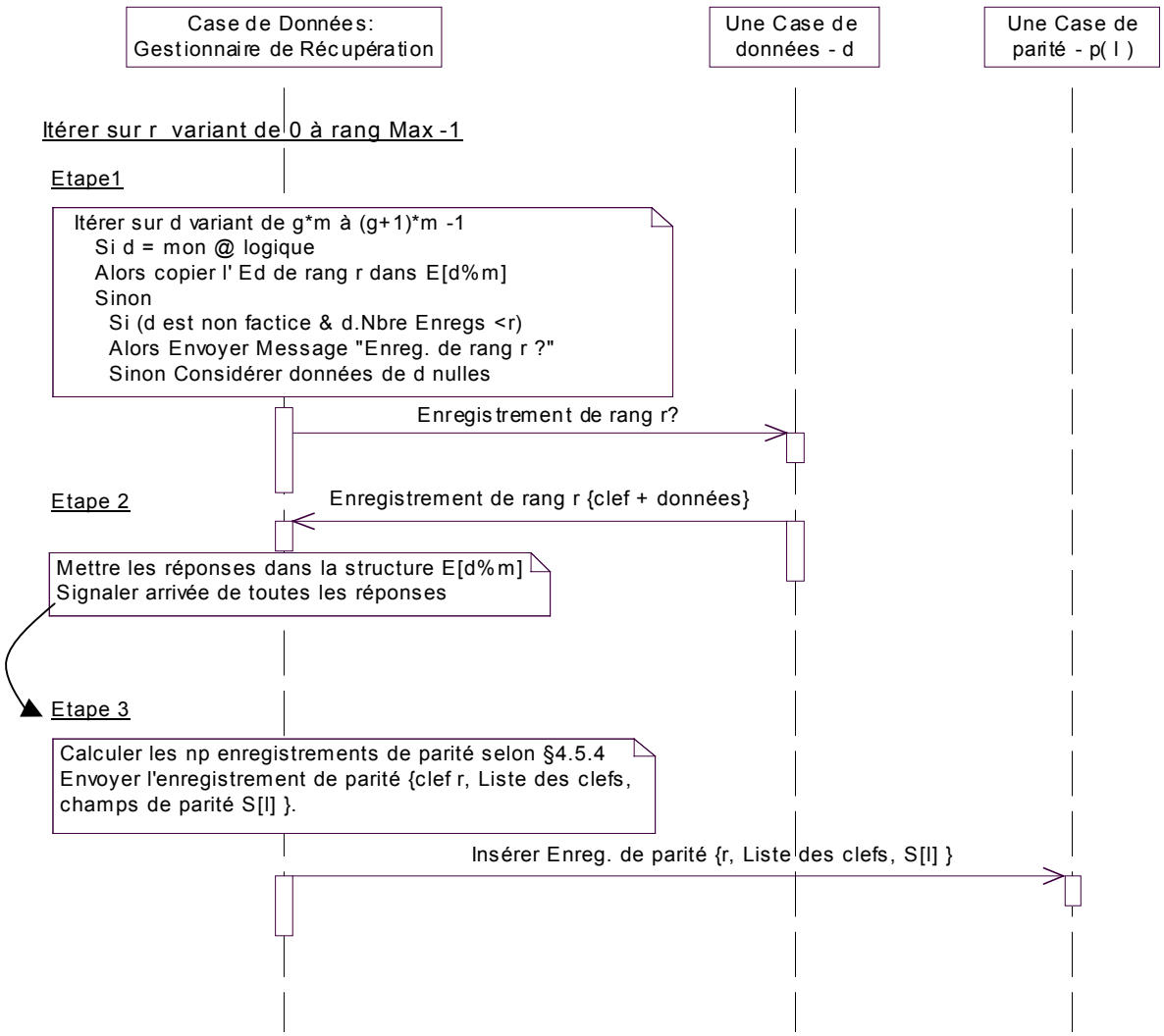


Figure 3.12: Scénario de récupération de l'ensemble des cases de parité

### 3.6.5.2 Case de Parité

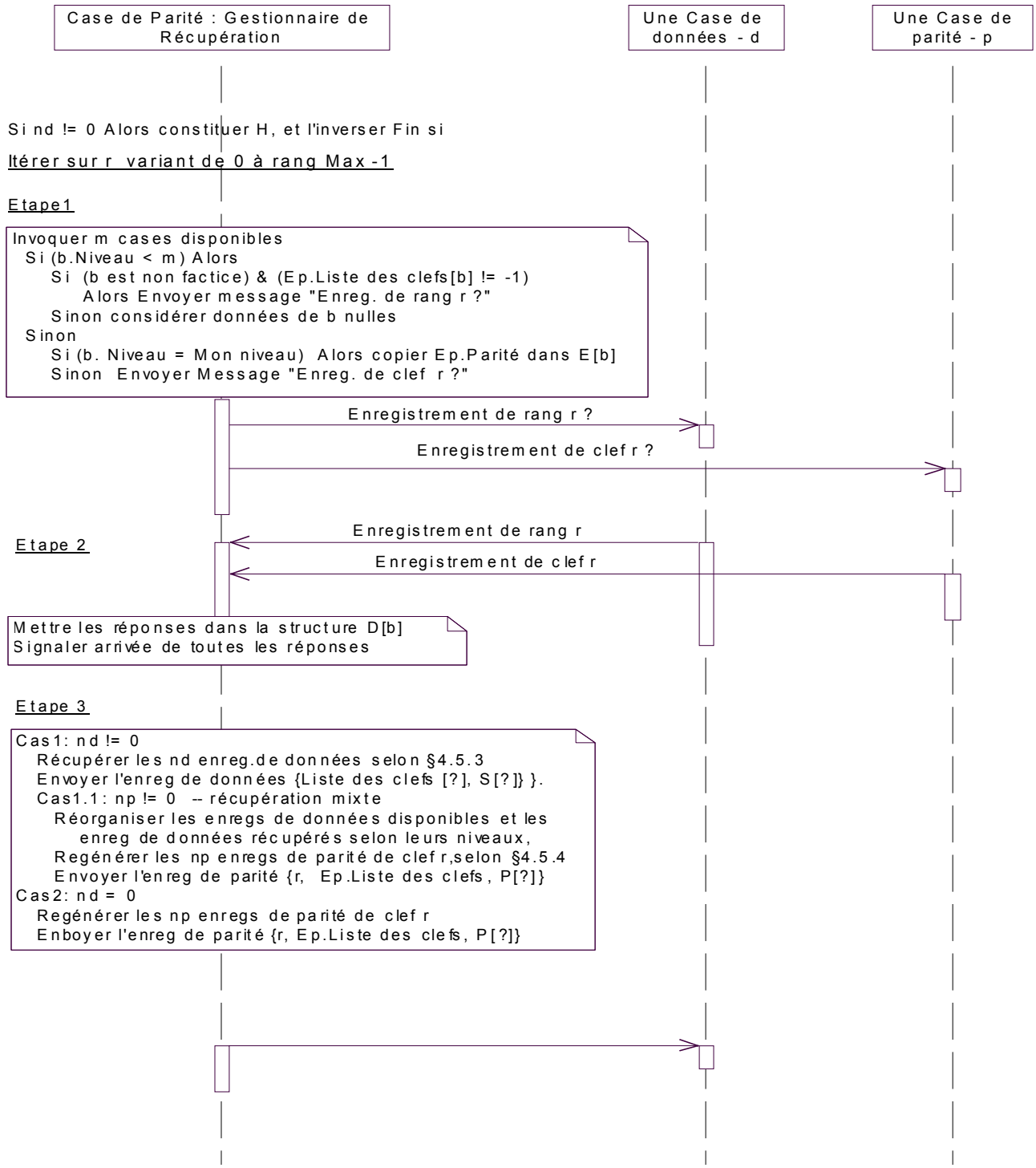


Figure 3.13 : Scénario de récupération mixte

## 3.7 Conclusion

Le chapitre suivant montre des évaluations des scénarios proposées, en particulier l'impact du scénario d'éclatement proposé, la comparaison des deux scénarios d'ajout de cases de parité, et enfin le scénario de récupération.

# 4 Mesures de performances

## 4.1 Introduction

Dans le présent chapitre, nous évaluons les performances des scénarios proposés, notamment,

- L'impact du scénario de transfert de mises à jour aux cases de parité suite à un éclatement, sur la création du fichier (§3.4.3).
- Comparaison des scénarios d'ajout d'une case de parité (§3.5).
- Evaluation du scénario de récupération (§3.6).

Les résultats feront l'objet d'analyses, suite desquelles des conclusions sont tirées.

L'environnement Expérimental, se compose de 6 machines Pentium 700 Mhz, chacune de 128 Mo de RAM, et exécutant Windows 2000. Les machines sont reliées par un réseau Ethernet 100 Mbps.

Tout au long du chapitre, nous adoptons la terminologie suivante,

*m* : nombre de cases de données dans un groupe,
*b* : capacité maximale d'une case de données,
*k* : niveau de disponibilité,
*t* : taille du champs Attributs d'un enregistrement de données, fixé à 100,
*N* : nombre d'enregistrements insérés dans le fichier.

En ce qui concerne le calcul de parité, nous travaillons sous GF($2^8$), et nous fixons *m* à 4.

## 4.2 Création du fichier LH*RS

L'expérimentation se déroule avec un client, trois serveurs de données et *k* serveurs de parité, tel que k $\in$ {0, 1, 2}. Nous fixons *b* à 10 000 et *N* à 25 000.

Le temps total de création d'un fichier sur plusieurs cases se mesure au niveau du client. Le client démarre son compteur de mesure avant le premier envoi. A la réception d'une clé *e* multiple de 2500, le client arrête son compteur, pour relever le temps d'insertion de *e* enregistrements.

Dans une première étape, nous évaluons le temps de création d'un fichier LH*$_{LH}$. Dans la première expérimentation, le client formule et envoie des requêtes d'insertion de 25 000 enregistrements de données, en 1.702 secondes, soit 0.068 ms/ requête. Nous notons un taux de perte négligeable de 0.024%. Pour avoir une idée sur le temps d'insertion d'un enregistrement de clé *e*, la case de données insérant *e* envoie un accusé au client si *e* est multiple de 2500.

Notons que les 25 000 requêtes d'insertion se dirigent vers la case de données 0, ce qui explique le nombre de messages d'ajustement d'image évoluant de 0 avant le premier éclatement à 8732 à la réception de l'accusé de la clé 25 000. Le temps moyen d'insertion d'un enregistrement de donnée est de 0.3 ms.

En passant à k égal à 1 et 2, le taux de perte est devenu de 15.052%. Ce qui nous contraint à ralentir le client de la manière suivante, le client n'envoie la requête *q* qu'après avoir reçu l'accusé de la requête *q-1*. A la réception de chaque accusé, le client ajuste son image, le nombre de messages d'ajustements de l'image du client est au nombre de 2. le client termine alors l'envoi de 25 000 requêtes d'insertion, et reçoit l'accusé de l'enregistrement de données de clé 25 000, en 15.712 s, 27.419s et 55.871s, respectivement pour *k* égal à 0, 1, 2.

Les résultats obtenus sont résumés dans la figure 4.1.

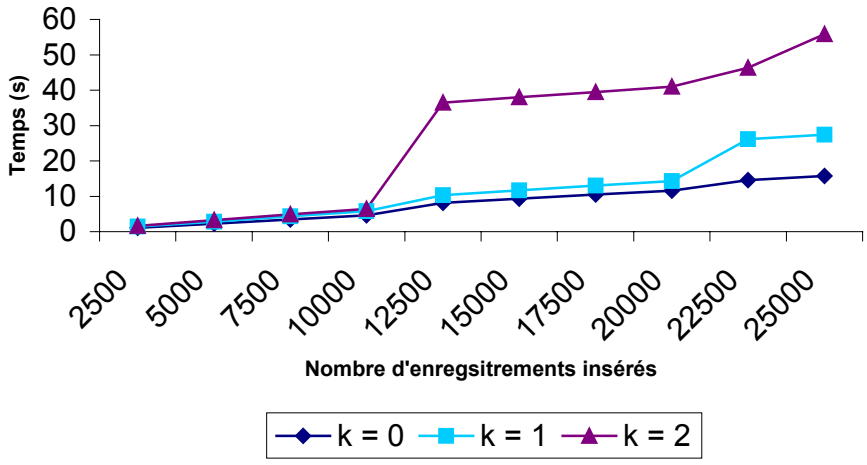


Figure 4.1: Variation du temps de création d'un fichier LH*$_{RS}$ en fonction de *k*

La figure 4.1, montre un temps de création de fichier croissant en fonction du nombre de cases de parité. Remarquons que dans l'intervalle [1,10000], les trois courbes se superposent, et la dégradation de performance par rapport à LH*$_{LH}$ est de 26.21%, 40.19%, respectivement pour *k* égal à 1 et 2. Par contre, après le premier éclatement, l'écart entre les courbes augmente, la latence est due aux connexions TCP établies entre la case de données qui éclate,

la nouvelle case de données et les *k* cases de parité. Une gestion plus sophistiquée des connexions TCP est alors requise. Sachant que, la case de parité traite un tampon TCP, de 5000 insertions et 10000 suppressions, en 0.921 secondes, soit 0.061 ms par opération de mise à jour.

## 4.3 Ajout de cases de parité

Nous avons mené des expérimentations, ayant pour objectif, la comparaison des deux scénarios proposés au chapitre précédent (§3.4). Rappelons qu'à l'issue des deux scénarios, une nouvelle case de parité est ajoutée à un groupe de cases de données. La différence entre les scénarios réside dans le protocole utilisé pour acheminer les messages.

Dans ce qui suit, nous évaluons le temps de création d'une nouvelle case de parité, en variant le nombre de cases de données dans le groupe.

- Fichier LH*RS sur une case de données

Dans une première étape, nous insérons un nombre d'enregistrements inférieur à *b*, puis nous procédons à l'ajout de deux cases de parité, conformément aux scénarios décrits en (§3.5.1) et (§3.5.2).

Les résultats obtenus sont résumés dans la figure 4.2.

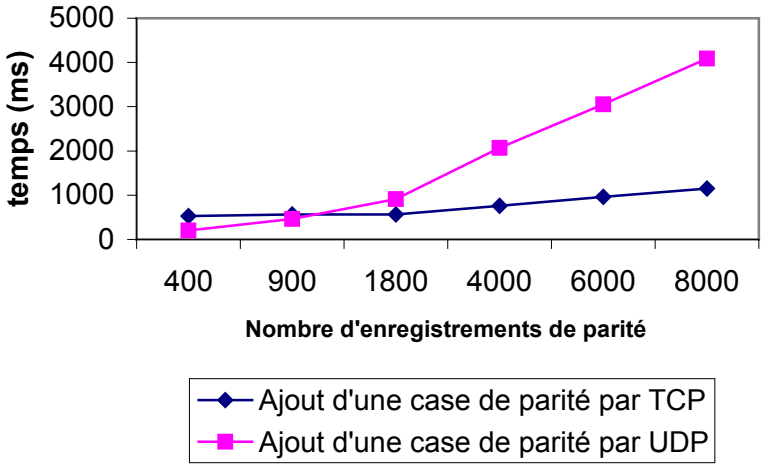


Figure 4.2: Création d'une case de parité
à partir d'une case de données

- Fichier LH*RS sur deux cases de données

Dans une première étape, nous insérons 2*b enregistrements de données, puis nous procédons à l'ajout de deux cases de parité, conformément aux scénarios décrits en (§3.5.1) et (§3.5.2).

Les résultats obtenus sont résumés dans la figure 4.3.

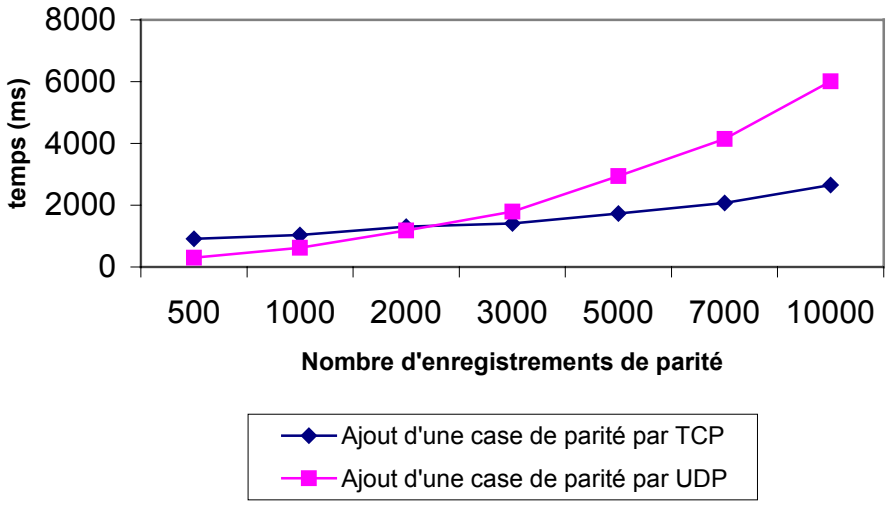


Figure 4.3: Création d'une case de parité
à partir de deux cases de données

- Fichier LH*RS sur trois cases de données

Dans une première étape, nous insérons 2.5*b enregistrements de données, puis nous procédons à l'ajout de deux cases de parité, conformément aux scénarios décrits en (§3.5.1) et (§3.5.2).

Les résultats obtenus sont résumés dans la figure 4.4.

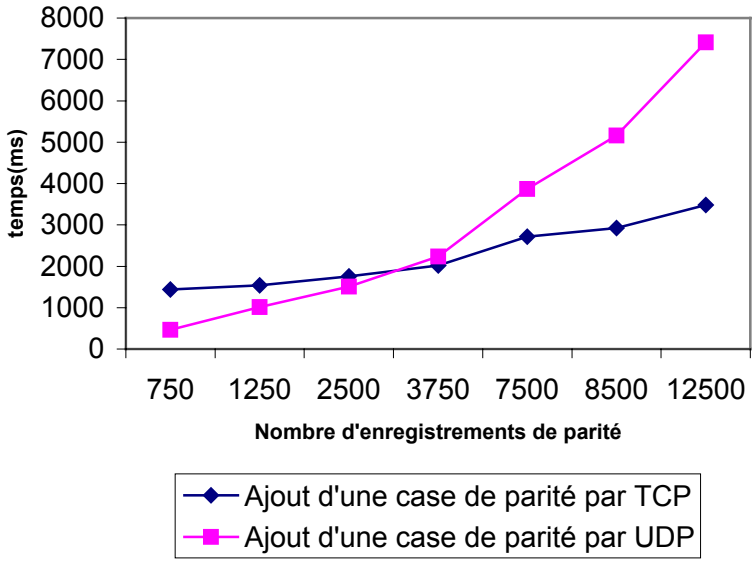


Figure 4.4: Création d'une case de parité
à partir de trois cases de données

Notons que,

- Dans toutes les expérimentations ci-dessous décrites, les deux courbes se croisent. Au-delà, d'un certain nombre d'enregistrements, les performances du scénario implantant le protocole TCP (§3.5.2) sont meilleures que celles du scénario basé sur le protocole UDP (§3.5.1).

- L'allure des résultats du scénario implantant le protocole TCP (§3.5.2) est plutôt linéaire.

Dans le but de valoir, le scénario de création d'une case de parité, décrit dans (§3.5.2), nous avons procédé à la mesure du temps au niveau des cases de données et de la nouvelle case de parité.

Nous avons choisi de créer le fichier sur deux cases de données, car dans ce cas chaque case de données, et également la nouvelle case de parité, comportera exactement *b* enregistrements de données.

Les résultats trouvés sont résumés dans le tableau ci-dessous,

| *b* | *Case de données 0 ( sec)* | *Case de données 1 (sec)* | *Traitement des tampons TCP (sec)* | *Temps de création de la case de parité(sec)* |
|---|---|---|---|---|
| 30000 | 0.871 | 0.871 | 3.725 | 6.109 |
| 40000 | 0.971 | 1.022 | 5.008 | 7.852 |
| 50000 | 1.161 | 1.202 | 6.380 | 9.824 |

Notons que,

- La connexion d'une case de données à une case de parité mobilise un processus de travail pendant un temps croissant en fonction de *b*, atteignant 1.161 secondes pour la préparation d'un tampon de 50000 enregistrements, connexion et transfert.

- Le temps moyen de mise à jour d'un enregistrement de parité est de 0.06 ms.

# 4.4 Récupération de cases de Parité

## 4.4.1 Expérimentations préliminaires

Nous avons procédé à une série d'expérimentations, tel que chaque expérimentation correspond à une capacité fixe b, et se déroule comme suit,

- En première étape, nous varions b sur l'ensemble {10 000, 15 000, 20 000, 30 000, 40 000, 50 000},
- En deuxième étape, nous insérons 2*b enregistrements de données dans un fichier LH*RS, de niveau de disponibilité k égal à 1. Dans chaque expérimentation le fichier se crée sur exactement deux cases de données, de b enregistrements chacune,
- Enfin, nous simulons l'échec de la case de parité et nous la récupérons.

Les résultats correspondant à cette série d'expérimentations, sont illustrés par la figure ci-dessous,

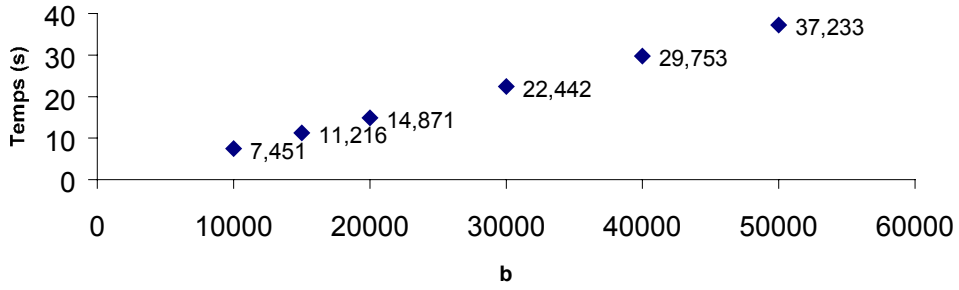


Figure 4.5 : Variation du temps de récupération d'une case de parité en fonction de b

Si nous procédons à la division du temps de récupération de la case de parité échouée par b, étant également le nombre d'enregistrements récupérés, nous trouvons une valeur constante égale à 0.7451 ms par enregistrement de parité récupéré. L'invariance du temps de temps de récupération d'un enregistrement de parité au nombre d'enregistrements à récupérer, étant b, prouve la scalabilié du scénario proposé en §3.6.

### 4.4.2 Récupération de k cases de parité

Dans ce qui suit, nous fixons b à 10 000. L'expérimentation se déroule comme suit, dans une première étape, nous créons un fichier LH*RS, de 20 000 enregistrements de données. Le fichier se crée exactement sur deux cases de données, chacune de 10 000 enregistrements de données. Puis, nous procédons à l'échec et la récupération des *k* cases de parité, sachant que, nous varions k, le nombre de cases de parité, de 1 à 4. Ceci dit, à chaque expérimentation, nous simulons l'échec de l'ensemble des k cases de parité.

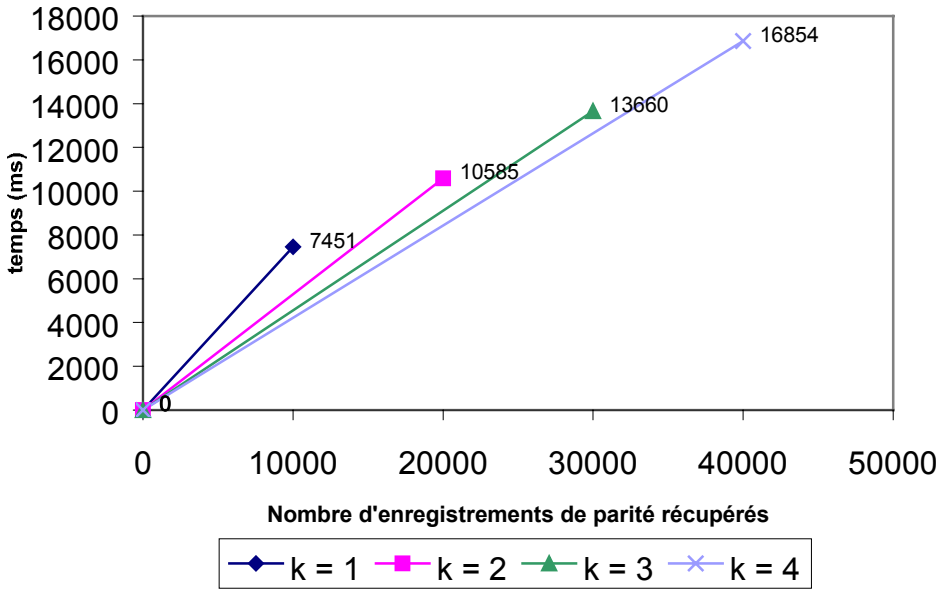


Figure 4.6 : Variation du temps de récupération de cases de parité en fonction du nombre de cases de parité à récupérer

La figure 4.6 montre un temps de 7.451 secondes pour la récupération, d'une totalité de 10000 enregistrements de parité. Nous remarquons, que le temps de récupération de $k$ cases de parité, est inférieur à $7.451 * k$ secondes. Ceci s'explique par une factorisation, des messages d'invocation des cases de données, le plus est du à la régénération des enregistrements de parité des $k$-$1$ autres cases de parité.

Remarquons que, quel que soit $k$ : le nombre de cases de parité à régénérer, la case de donnée gestionnaire de récupération exécute 10 000 itérations. Dans ce qui suit, nous procédons à une analyse numérique des résultats obtenus,

- Pour $k$ égal à 1, une itération est de 0.7451 ms.
- Pour $k$ égal à 2, une itération est de 1.0585 ms. Ceci implique que le traitement inhérent au deuxième enregistrement de parité, est de 0.3134 ms ( = 1.0585 ms – 0.7451 ms).
- Pour $k$ égal à 3, une itération est de 1.3660 ms. Ceci implique que deux enregistrements de parité nécessite 0.6209 ms ( = 1.3660 ms – 0.7451 ms), remarquons que c'est presque égal au double de 0.3134 ms.
- Pour $k$ égal à 4, une itération est de 1.6854 ms. Ceci implique que deux enregistrements de parité nécessite 0.9403 ms ( = 1.6854 ms – 0.7451 ms), remarquons que c'est presque égal au triple de 0.3134 ms.

La figure 4.7, montre le temps de récupération d'un enregistrement de parité

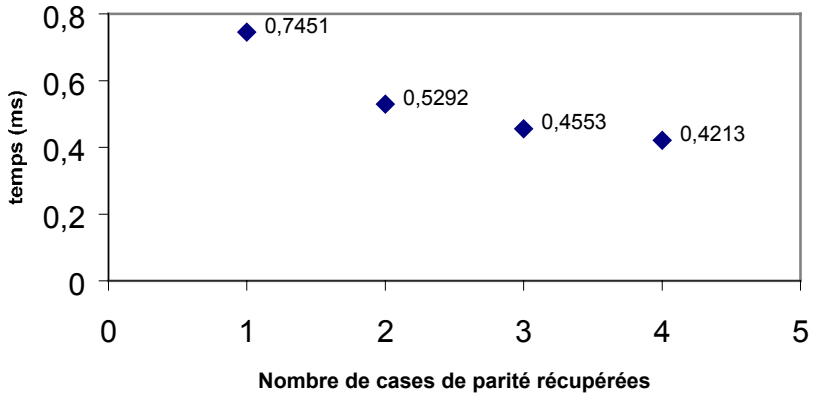


Figure 4.7: Variation du temps de récupération d'un enregistrement de parité en fonction du nombre de cases de parité récupérées

La figure 4.7 montre que, plus le nombre de cases de parité à récupérer augmente, le temps de récupération d'un enregistrement, nous fait l'illusion qu'il diminue, alors que, c'est la factorisation du nombre de messages d'invocation, qui fait que le temps récupération de k enregistrements soit inférieur à $0.7451 * k$ ms.

Il est à remarquer, que nous pouvons substituer le scénario de récupération d'une case de parité, par celui le scénario de création d'une case de parité (§3.5.2). Dans ce sens, nous avons mesuré le temps de création d'une case de parité, à partir de deux cases de données, nous avons trouvé un temps égal 2653 ms. Ceci dit, la re-génération de $k$ cases de parité serait $k$* 2653 ms.

Dès lors, les mesures de performance remettent en cause le scénario proposé en §3.6.5.1, en faveur d'un nouveau scénario de récupération de $x$ cases de parité. Ce scénario se présentera comme suit : suite à la détection d'un échec au niveau de $x$ cases de parité, le coordinateur envoie x messages d' « auto-génération » aux $x$ cases de parité remplaçant celles en échec. Il est à noter, que ces cases de parité se connecteront simultanément aux cases de données du groupe. Différentes stratégies d'optimisation du traitement peuvent être proposées, notamment celle jouant sur l'ordre des connexions. Chaque case de parité se connectera dans un ordre précis aux cases de données, d'une part pour accélérer le traitement, et d'autre part pour alléger la synchronisation et le temps d'attente, car toute source de latence, augmente la divergence entre les cases de données et celles de parité correspondantes.

## 4.5 Récupération de cases de données

Comme la création du fichier, ne garantie pas la cohérence des cases de parité aux cases de données. Pour effectuer la récupération d'une case de données, nous avons procédé de la manière suivante,

- En première étape, un fichier LH*RS, de niveau de disponibilité k égal à 0, est crée,
- En deuxième étape, nous ajoutons une case de parité au fichier. Ce dernier devient de niveau de disponibilité k égal à 1, et peut survivre à l 'échec d'une case de données.

Nous avons mené une série d'expérimentations, tel que chaque expérimentation correspond à une capacité fixe b, et se déroule comme suit,

- En première étape, nous varions b sur l'ensemble {10 000, 20 000, 30 000, 40 000, 50 000},
- En deuxième étape, nous insérons 2*b enregistrements de données dans un fichier LH*RS, de niveau de disponibilité k égal à 1. Dans chaque expérimentation le fichier se crée sur exactement deux cases de données, de b enregistrements chacune,
- Enfin, nous simulons l'échec d'une case de données et nous la récupérons.

Les résultats correspondant à cette série d'expérimentations, sont illustrés par la figure ci-dessous,

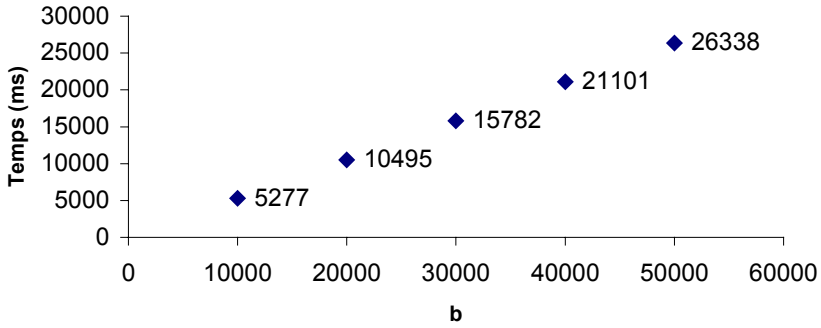


Figure 4.8: Variation du temps de récupération d'une case de données en fonction de b

De futures expérimentations doivent porter sur la récupération de plusieurs cases de données, qui sont expectées similaires à celles de la récupération de plusieurs cases de parité (§4.4.2).

## 4.6 Conclusion & Limites des expérimentations

Tout au long de ce chapitre, nous avons mis en œuvre des séries d'expérimentations, qui nous ont permis d'évaluer les scénarios proposés, et conclure que,

- Les expérimentations ont montré des sources de latence, au moment de l'éclatement, une meilleure gestion des connexions TCP établies avec les cases de parité au moment d'un éclatement, serait d'un grand apport au prototype.

- Le scénario de récupération de cases de parité, doit être substitué par celui de régénération d'une case de parité par le protocole TCP.

- Les messages de mise à jour des cases de données vers les cases de parité doivent être acheminées par un protocole fiable, pour pouvoir procéder à des récupérations.

# Conclusion & Perspectives

Dans ce mémoire, nous avons

- Survolé différentes alternatives pour rendre un système de stockage fiable,
- Approfondi la structure de données distribuée et à haute disponibilité scalable : LH*RS [Litwin & Schwartz, 2000],
- Proposé des scénarios relatifs à la mise à jour des cases de parité lors d'un éclatement, l'ajout d'une case de parité pour augmenter la disponibilité du fichier, et un scénario de récupération général,
- Enfin, nous avons testé les performances des scénarios proposés.

Les travaux futurs portent sur :

- Une mise en œuvre plus élaborée de LH*RS,
- Recherche d'autres codes dont le calcul de parité est moins contraignant.

# Bibliographie


[Bennour, 2000] F. Bennour, *Un Gestionnaire de Structures Distribuées et Scalables pour les multiordinateurs Windows : Fragmentation par Hachage*, Rapport de thèse, Université Paris Dauphine,2000.

[Blaum et al., 1994] M. Blaum, J. Brady, J. Bruck & J. Menon, *EVENODD : An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures*, IEEE 1994.

[Blomer et al., 1995 ] 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.

[DeWitt & Gray, 1992] D. DeWitt & J. Gray, *Parallel Database Systems : The Future of High Performance Database Systems*, Communication of ACM, June 1992, Vol. 35, N°6.

[Hellerstein et al., 1994] L. Hellerstein, G.A. Gibson, R.M. Karp, R.H. Katz & D.A. Patterson, *Coding Techniquesfor handling Failures in LargeDisk Arrays*, Algorithmica, 1994, 12, pp.182-208.

[Karlson et al., 1996] J. karlson, W. Litwin & T.Risch, LH*LH: *A Scalable high performance data structure for switched multicomputers*, EDBT 96, Springer Verlag.

[Lin et al., 1983] S. Lin & D.J. Costello, *Error Control Coding: Fundamentals and Application*, Prentice Hall, 1983.

[Lindberg, 1997] 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

[Litwin et al., 1994] 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.

[Litwin & Neimat, 1996] W. Litwin & A.M. Neimat, *High Availability LH* Schemes with Mirroring*, Intl. Conf on Cooperating systems, Brussels, IEEE Press 1996.

[Litwin et Risch, 1997] W. Litwin & T. Risch, *LH*g: A High-Availability Scalable data Structure by Record Grouping*, Res. Rep. U. Paris9 & U. Linkoping, 1997.

[Litwin et al, 1997] 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.

[Litwin et al., 1998] W. Litwin, J. Menon & T. Risch, *LH* with Scalable Availability*, IBM Almaden research report RJ 10121 (91937), May 1998.

[Litwin et al., 1999] 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.

[Litwin & Schwarz, 2000] 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.

[Ljungström, 2000] M. Ljungström, *Implementing LH*RS : a Scalable Distributed Highly-Availability Data Structure*, Master Thesis, Feb. 2000, CS Dep., U. Linkoping, Suede.

[Mogi et al., 1996] 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.

[Patterson et al., 1988] Patterson, G. Gibson & R.H. Katz, *A Case for Redundant Arrays of Inexpensive Disks*, Proc. of ACM SIGMOD Conf, pp.109-106, June 1988.

[Plank, 1997] 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, http://www.cs.utk.edu/~plank/plank/papers/cs.96.332.html

[Rabin, 1989] 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.

[Rizzo, 1996] L. Rizzo, *On the Feasability of Software FEC*, DEIT Tech. Report LR970131,1996, http://www.iet.unipi.it/~luigi/softfec.ps.

[Rizzo, 1997] L. Rizzo, *Effective Erasure Codes for reliable computer communication protocols*, ACM Computer Communication Review, Vol.27, Apr. 97, pp. 24-36.

[SDDS] http://192.134.119.81/SDDS-bibliographie.html

[Vingralek et al., 1994] R.Vingralek, Y.Breitbart & G.Weikum, *Distributed File Organisation with Scalable Cost/Performance*, ACM SIGMOD Int. Conf. on Management of Data, 1994.

[Vingralek et al., 1995] R.Vingralek, Y. Breitbart & G. Weikum, *SNOWBALL : Scalable Storage on Networks of Worstations with Balanced Load*, http://www.bell-labs.com/user/rvingral/publications.html

[Xu et al., 1999] 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.