UNIVERSITE PARIS DAUPHINE

DFR SCIENCES DES ORGANISATIONS

Centre de Recherche en Informatique Appliquée

THÈSE

Signatures Algébriques dans la Gestion de Structures de Données Distribuées et Scalables

Pour l'obtention du titre de

Docteur en Informatique

(Arrêté du 25 Avril 2002)

Présentée et soutenue le 22 Juin 2006

Riad MOKADEM

Composition du Jury

Directeur de Thèse :

Witold LITWIN

Professeur à l'Université Paris IX Dauphine.

Président de Jury :

Phillipe RIGAUX

Professeur à l'Université Paris IX Dauphine.

Rapporteurs :

Patrick VALDURIEZ

Professeur Directeur de Recherche à l'INRIA Nantes.

Thomas SCHWARZ

Professeur à l'Université Santa Clara. USA.

Suffragant :

Tore Risch

Professeur à l'Université UPPSALA. Suède.

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

Remerciements

Mes remerciements reviennent d'abord à mon directeur de Thèse, le Professeur Witold Litwin, Directeur du CERIA (Centre de d'Etude et de Recherche en Informatique Appliquée) à l'Université Paris XI Dauphine, pour avoir supervisé ces travaux. Sa grande disponibilité constante, sa persévérance ainsi que son souci du détail m'ont beaucoup aidée tout le long de ce travail. Merci pour la pertinence des jugements ainsi que pour les encouragements.

Je remercie vivement le Professeur Philippe Rigaux qui m'a fait l'honneur de présider le jury de cette thèse. Ses suggestions durant la pré soutenance m'ont été très utiles.
Je remercie chaleureusement le Professeur Thomas Schwarz, Professeur à l'Université de Santa Clara, aux Etats Unis d'Amériques. Merci pour les conseils éclairés et les discussions fructueuses
Je remercie également le professeur Patrick Valduriez, Professeur à l'institut de Recherche en Informatique de Nantes (IRIN), d'avoir accepter d'être rapporteur de mon travail.
Je remercie le Professeur Tore Risch, Directeur du Laboratoire de base de données UDBL à l'Université de Uppsala – Suède pour son interet à nos travaux..

Je remercie également les coordinateurs de Microsoft Corp, particulièrement Dr Jim Gray, pour l'intérêt qu'ils portent à mon travail. Mes remerciement aussi pour Monsieur Xech, responsable des Relations Universitaire à Microsoft Research pour son engagement auprès de sa direction pour le soutient financier, matériel et logiciel de Microsoft qui nous ont permis de poursuivre cette Thèse.

Je remercie le Professeur Lecroq, Professeur à l'Université de Rouen d'avoir bien voulu répondre à mes questions

Je remercie sincèrement l'équipe du CERIA, Rym, Fatma, Djelloul, Hanafi, Janine, Sylvie et tout particulièrement Soror, pour leurs discussions, bonne humeur, remarques et disponibilité tout au long de ce travail. Mes remerciements vont également à Monsieur Aly Wane Diène pour sa disponibilité surtout au début de ma thèse.

J'exprime mes remerciements également à Monsieur Gérard Lévy, Professeur à l'Université Paris XI Dauphine, Mme Jacqueline De Labruslerie, directrice de l'UFR de Gestion à l'Université Paris

Dauphine ainsi que Monsieur Zeggour Djamel, Professeur à l'INI d'Alger pour leur collaborations et conseils judicieux tout au long de ma Thèse.

Toute ma reconnaissance va également à mes très chers parents, Boualem et Fatima ainsi que mes frères et sœur, Fares, Nadir, Manel et Massil. Merci pour leur soutien et leurs encouragements.

Mes remerciement également à mes amis (es) et tous particuliérement Hamid et sa petite famille, Rachid ainsi que Athmane et Wassila qui ont bien voulu relire ce manuscrit.

Les travaux de cette Thèse on été partiellement financés par le projet CEE IST ICONS ainsi qu'un contrat de recherche du programme "Innovation Excellence" de MS Research. Nous citerons également un don de MS Research pour Université Paris Dauphine et le projet eGov de la CEE.

Résumé

Les deux dernières décennies ont été marquées par l'apparition de nouveaux concepts architecturaux entraînant une évolution vers les systèmes distribués. C'est une conséquence de l'augmentation de la capacité de stockage des mémoires et de calcul et de l'arrivée de réseaux à haut débit, notamment locaux à 1Gb/s. La tendance dominante est le développement de nouveaux systèmes, dits d'abord: *multi-ordinateur*, *Réseau de Stations de Travail* et plus récemment, « *Peer-to-Peer Computing* » ou « *Grid Computing* ». Afin de tirer le meilleur profit des potentialités offertes, de nouvelles structures de données spécifiques aux données réparties sont nécessaires.

Dans ce contexte, Les Structures de Données Distribuées et Scalables (SDDS) sont une nouvelle classe de structures introduites spécifiquement pour la gestion de fichiers sur un multi ordinateur. Un fichier SDDS peut s'étendre dynamiquement, au fur et à mesure des insertions, d'un seul site de stockage à tout nombre de sites interconnectés disponibles en pratique. Les algorithmes d'adressages d'une SDDS sont conçus spécifiquement pour être scalables, notamment par absence d'un répertoire ou index central. La répartition de données est transparente pour l'application. Les données manipulées peuvent être entièrement en RAM distribuée afin d'être accessibles bien plus vite qu'à partir des disques. Plusieurs SDDS ont été proposées. Les plus connues sont celles basées sur le hachage, celui linéaire (LH*) notamment, et celles utilisant le partitionnement par intervalle (RP*). Un prototype appelé SDDS-2000a été construit vers l'année 2000 au CERIA pour expérimenter avec les SDDS sur les réseaux locaux des PC sous Windows. Dans ce système, on retrouve les fonctions de base de gestion de données telles que la création de fichiers, l'insertion d'enregistrements ou encore la possibilité de requêtes parallèles.

En se basant sur SDDS-2000, notre Thèse a pour objectif la conception et l'implantation de nouvelles fonctions pour celui ci. Ces fonctions sont destinées à la sauvegarde de données sur le disque, un traitement plus efficace de mises à jour, le traitement de concurrence ainsi que celui de la recherche par le contenu (scans). Enfin, pour mieux répondre au contexte P2P, il nous fallait introduire une certaine protection de données stockées, au moins contre une découverte accidentelle de leurs valeurs. Ceci nous a conduit au problème intéressant de recherche de données par l'exploration directe de leur contenu encodé, sans décodage local.

Nous avons basé l'ensemble de nos fonctions sur une technique nouvelle dite de signatures algébriques. Nous détaillons la théorie et notre pratique de signatures algébriques tout au long de cette

Thèse. Ainsi, une sauvegarde sur disque n'écrit que les parties de la RAM modifiées depuis la dernière sauvegarde. Le contrôle de concurrence est optimiste, sans verrouillage, pour de meilleures performances d'accès. L'enregistrement mis à jour n'est envoyé au serveur que si la donnée est réellement modifiée. Puis, les données stockées sont suffisamment encodées pour rendre impossible toute découverte accidentelle de leurs valeurs réelles sur les serveurs. Nous les encodons à l'aide d'une variante de signatures algébriques, les signatures cumulatives. Notre encodage possède notamment des propriétés accélérant diverses recherches de chaînes de caractères, par rapport à celles explorant les mêmes données sans encodage. D'une manière un peu surprenante, certaines recherches se révèlent expérimentalement plus rapides que par des algorithmes fondamentaux bien connus, tels que celui de Karp-Rabin.

Nous présentons des mesures de performance prouvant l'efficacité de notre approche. Notre système, appelé SDS-2005, a été dès lors annoncé sur DbWorld. Il est disponible sur le site du CERIA pour les téléchargements non commerciaux. Les détails de nos travaux ont fait l'objet de cinq publications dans des conférences internationales [LMS03, LMS05a, LMS05b, M06, LMRS06]. Notre prototype a également été montré à de nombreux visiteurs chercheurs. Il a fait l'objet d'une démonstration vidéo, diffusée notamment à Microsoft Research (Montain View, USA) et d'une présentation lors des journées académiques Microsoft.

Dans notre mémoire, nous présentons d'abord l'état de l'art sur les SDDSs, en se basant sur celui de systèmes de fichiers distribués. Puis nous discutons l'architecture système de SDDS-2005. Celle-ci emploie notamment des structures de données spécifiques pour RAM, ainsi que des processus légers qui gèrent les traitements répartis à travers des files d'attente asynchrones. On présente ensuite le concept de signatures algébriques. Puis on détaille l'usage pour la sauvegarde d'un fichier SDDS et la mise à jour d'enregistrements. Nous discutons ensuite les signatures cumulatives. On décrit l'encodage de nos enregistrements. On présente les différents types de recherche par contenu non-clé (scans) dans notre système notamment la recherche par le préfixe et celle partielle d'une chaîne de caractère (ang pattern matching ou string search…) à travers plusieurs algorithmes alternatifs. Nous présentons un nouvel algorithme dit par *n*-Gramme semblant particulièrement simple d'usage et rapide On décrit aussi la recherche du plus grand préfixe et de la plus grande chaîne commune. Nous montrons que les signatures cumulatives sont particulièrement efficaces pour la recherche de longues chaînes telles que les images, les empreintes, les codes DNA…En réflexion sur les perspectives, on discute l'utilisation de ces signatures pour la compression différentielles lors des mises à jour distribuées des données ainsi que la protection contre la corruption silencieuse de données stockées.

Puis nous discutons l'analyse expérimentale de notre système. Les mesures montrent la scalabilité de notre système ainsi que les temps d'exécution de nos différentes fonctions. On finit par des conclusions, perspectives et les références bibliographiques. Les annexes montrent nos principales publications (pour la convenance des membres anglophones de notre jury tout particulièrement). On y montre aussi la description de l'interface offerte aux applications par SDDS-2005, annoncée sur DbWorld.

Mots clés: *Multi ordinateur, scalabilité, Structures de Données Distribuées et Scalables, Signatures algébriques, Traitement parallèle*, Recherche de chaînes.

Abstract

Recent years saw emergence of new architectures, involving multiple computers. New concepts were proposed. Among most popular are those of a multicomputer or of a Network of Worksattion and more recently, of Peer to Peer and Grid Computing.

This thesis consists on the design, implementation and performance measurements of a prototype SDDS manager, called SDDS-2005. It manages key based ordered files in distributed RAM of Windows machines forming a grid or P2P network. Our scheme can backup the RAM on each storage node onto the local disk. Our goal is to write only the data that has changed since the last backup. We interest also to update records and non key search (scans). Their common denominator was some application of the properties of new signature scheme based that we call algebraic signatures, which are useful in this context. Ones needs then to find only the areas that changed in the bucket since the last buckup. Our signature based scheme for updating records at the SDDS client should prove its advantages in client-server based database systems in general. It holds the promise of interesting possibilities for transactional concurrency control, beyond the mere avoidance of lost updates. We also update only data have been changed because of the using the algebraic signatures.

Also, partly pre-computed algebraic signature of a string encodes each symbol by its cumulative signatures. They protect the SDDS data against incidental viewing by an unauthorized server's administrator. The method appears attractive, it does not amply any storage overhead. It is also completly transparent for servers and occurs in client. Next, our cheme provide fast string search (match) directly on encoded data at the SDDS servers. They appear an alternative to known Karp-Rabin type schemes. Scans can explore the storage nodes in parallel. They match the records by entire non-key content or by its substring, prefix, longest common prefix or longest common string. The search complexity is almost O (1) for prefix search. One may use them also to detect and localize the silent corruption. These features should be of interest to P2P and grid computing.

Then, we propose novel string search algorithm called *n*-Gramme search. It also appears then among the fastest known, e.g, probably often the faster one we know. It cost only a small fraction of existing records match, especially for larger strings search.

The experiments prove high efficiency of our implementation. Our buckup scheme is substantially more efficient with the algebraic signatures. The signature calculus is itself substantially faster, the gain being about 30 %. Also, experiments prove that our cumulative pre-computing notably accelerates the string searchs which are faster than the partial one, at the expense of higher encoding/decoding

overhead. They are new alternatives to known Karp-Rabin type schemes, and likely to be usually faster. The speed of string matches opens interesting perspectives for the popular join, group-by, rollup, and cube database operations.

Our work has been subject of five publications in international conferences [LMS03, LMS05a, LMS05b, ML06, l&al06]. For convenience, we have included the latest publications. Also, the package termed SDDS-2005 is available for non-commercial use at http://ceria.dauphine.fr/. It builds up on earlier versions of the prototype, a cumulative effort of several folks and *n*-Gramme algorithm implementation. We have also presented our proposed prototype, SDDS-2005, at the *Microsoft Research Academic Days 2006*.

Key words: *Multicomputer, scalability, scalable distributed data structures, parallel processing, algebraic signatures, string Matching*

Extended Abstract

Thesis Goal

This thesis focuses on the design, implementation and performance measurements of a new prototype SDDS manager, called SDDS-2005. It manages scalable key based ordered files in distributed RAM of Windows machines forming a grid or P2P network. The SDDS-2005 files are the scalable distributed data structures (SDDSs) of RP * type. The data are basically stored in the distributed RAM, for faster access than to the traditional disk-based structures.

Our work extends a previous version of the prototype, termed SDDS-2000. We have added several functions. Their common denominator was some application of the properties of the algebraic signatures, [LS03, LMS03]. We have analyzed the design of the algorithms underlying each function. We pruned various design options, to choose the variants that seemed best for our goal. We have implemented the final design into the prototype. We have performed the theoretical and experimental validation of our choices. Finally, we have brought the prototype to the state sufficient to be the "proof-of-the-concept" available to the research community worldwide. For this purpose we made the prototype available for download at CERIA Web site, and annonced it at DbWorld.

Our first function was a backup scheme storing the RAM data file (the local SDDS bucket) on each storage node onto the local disk. The reasons were the volatility of the RAM and its limited capacity. Our goal was to write only the data that has changed since the last backup. The algebraic signature calculus that we have developed for this purpose provided substantial gains with respect to the naïve backup scheme. The signature calculus is itself substantially faster and enough to remain under 10 % of the disk write time. The gain is about 30 %. Our buckup scheme is substantially more efficient with the algebraic signatures.

Using the algebraic signatures, we have also added the concurrent update management. We update only data have been changed. Our signature based scheme for updating records at the SDDS client should prove its advantages in client-server based database systems in general. It holds the promise of interesting possibilities for transactional concurrency control, beyond the mere avoidance of lost updates.

Our next goal was to encode the stored data secure so that their value can not be incidentally discovered at the server by an unauthorized administrator or user. This property is important in P2P environment. It requires that one may manipulate the stored data in the encoded form at the server. We have offered this capability using a specific type of algebraic signatures termed the cumulative ones. A cumulative algebraic signature encodes each symbol with the signature of the string prefix ending with the symbol. It does not imply any storage overhead. It creates only small processing overhead. We have shown interesting computational properties of this technique.

Our other goal was various distributed string search capabilities, i.e., the pattern matching scans. The algebraic signatures proved useful in this context as well. On the one hand, we allow for the parallel distributed search using the pattern signature only. This makes the communication between the SDDS client and server more efficient. No actual pattern is sent on the network which is obviously interesting for the security. We do it in the way somehow similar to the well-known Karp-Rabin algorithm. We have also defined algorithms for various string searches over the encoded data. Our algorithms match the records by entire non-key content or by any substring (the pattern), the prefix, the longest common prefix or the longest common string with respect to the pattern. We process these without the local decoding. The operations prove very fast, usually faster, or much faster, than on the decoded data. This surprising result is due to the attractive properties of the cumulative algebraic signatures.

To validate our proposals, we have made various performance measurements and simulations. Our platform consisted of six Pentium VI 1.8 GHz machines with Windows 2000. Each site had minimum of 750MB of RAM. The machines were linked by a 1 GB/s Ethernet. The experiments showed high efficiency of our techniques. Our string search algorithms are now alternatives to well-known ones, the Karp-Rabin and Boyer-Moore especially.

We believe that our work shows that SDDSs represent a major step towards the operational use of multi-computers. Among the modern application that may benefit from this new technology are those on the grid and P2P configurations. There are also the Network Attached Storage (NAS) configurations. Likewise, SDDSs should prove useful for the DBMSs, the Web or multimedia servers. Generally, they should prove useful to any service needing the efficient access to large scalable and distributed data collections.

Key words: *Multicomputer, Scalability, Scalable Distributed Data Structures, Parallel processing, B-Tree. Algebraic signatures, String matching, Text Search*

Publications

We have published five papers about our work in international conferences [M03, LMS05a, LMS05b, ML06, L&al06a]. For convenience, we have included the latest publications. Also, as we said, our prototype is available for non-commercial use at http://ceria.dauphine.fr/. It builds up on earlier versions of the prototype, a cumulative effort of several folks and *n*-Gramme algorithm implementation. We have also presented our proposed prototype, SDDS-2005, at the *Microsoft Research Academic Days 2006*.

In [LMS03], we have presented our backup scheme in SDDS System to store RAM on each storage node onto the local disk. We write to disk only the data that has changed since the last backup. We experiment for this purpose with the algebraic signatures, more efficient than, e.g., the well-known SHA-1 standard. We present our architecture and design choices.

In [LMS05a] & [LMS05b], we describe the encoding/ decoding the data in SDDS-2005. We present there the algebraic cumulative signatures. As we mentioned, these protect also the SDDS data against incidental viewing by an unauthorized server's administrator. We explore them for the parallel scans of the storage nodes searching strings. We show how to match the records by entire non-key content or by a substring, a prefix, the longest common prefix or the longest common string, with respect to the pattern. We also signal further applications like the detection and localization of the silent corruption and the delta compression.

In [ML06], we present our update scheme using algebraic signatures. They allow efficiently for the optimistic concurrency control. Also, they serve efficiently for non-key and partial match searches. We compare our experiment results to Karp-Rabin's one.

In [L&al06a], we present our new *n*-Gramme algorithm search particularly efficient for longer patterns in the context of the database search. It seems faster than the prominent algorithms.

Thesis Outline

We divided the Dissertation into ten chapters. The "*Introduction*", discusses the basic issues and the motivation for the thesis work. In Chapter 2, "*Les Structures de Données Distribuées et Scalables*", we cover the basic features of networked data storage Systems. We also present the Scalable and Distributed Data Structures (SDDS). In Chapter 3, "*Prototype SDDS-2005*" describes SDDS-2005 architecture and other basic features of our prototype SDDS-2005. Chapter 4, "*Signatures Algébriques*", covers the theory of the algebraic signatures. Chapter 5, "*Signatures Algébriques pour la Sauvegarde de Données dans SDDS-2005*", describes our buckup scheme.

Next, Chapter 6, " *Signatures Algébriques pour la Mise à Jour de Données dans SDDS-2005s*", describe our update scheme. Chapter 7, "*Signatures Algébriques Cumulatives*", present the cumulative algebraic signatures and our application of their properties. Chapter 8, "*Mesures de Performances*", reports the results of the conducted experiments to prove the efficiency of our prototype. Finally, the last chapter "*Conclusion and Perspectives*" concludes the thesis dissertation and gives some future research directions.

The Thesis contains also two appendixes *A and B*. Appendix A, "SDDS-2005 V 1.0 Interface Functions", presents the internal processing of SDDS-2005 commands. We also present the Setup SDDS-2005 and important remarks about it. In Appendix B, we included our research papers for the convenience of our English speaking readers.

Table des Matières

Table des Figures

Algorithmes et Tableaux

Chapitre

1 INTRODUCTION

1.1 Motivations

Les réseaux de PCs et de Station de Travail sont devenus omniprésents dans les entreprises et pratiquement dans toutes les institutions. Ils offrent des capacités de stockage et des puissances de calcul non atteignables par les machines actuelles les plus puissantes [G93] [RR00]. L'arrivée des réseaux à haut débit, 1Gb/s, et la baisse continue de coûts ont facilité cette évolution. De nouveaux concepts architecturaux ont vu le jour. On citera notamment le concept de *multi-ordinateur* visant à combiner la puissance d'un grand nombre de machines. Afin de tirer profit de ces potentialités offertes, de nouvelles structures de données sont nécessaires. Dans ce contexte, les Structure de Données Distribuées et Scalables (SDDS) ont été proposées, tout particulièrement pour des fichiers manipulés en RAM distribuée dans un environnement de multi-ordinateur. Un prototype initial SDDS-2000 [D01] a été conçu pour cette approche. Il a été conçu pour les réseaux de station de travail Windows 2000 et Windows NT. Il permet de répartir dynamiquement un fichier SDDS sur la mémoire vive disponible sur un ensemble de PCs Windows. Le fichier peut potentiellement s'étendre progressivement sur des milliers d'ordinateurs.

A la suite de cette étude préliminaire, notre choix s'est porté sur la conception suivante :

- Stockage et recherche à travers la grille de PC : Les données et méta données sont au niveau de serveurs distribués permettant ainsi de garder notre architecture (architecture de multi ordinateur). Nous verrons par la suite qu'elle procure des capacités de passage à l'échelle (scalabilité).
- Approche logicielle totalement distribuée : Cette approche permet de procurer une transparence de gestion des opérations de recherche, mise à jour et de stockage traités en des

temps très réduits. Cela est possible grâce à l'utilisation de différentes techniques au niveau des serveurs distribués. Des requêtes parallèles sont aussi utilisées.

Dans ce cadre, nous avons étudié le prototype initial puis étendu ces fonctionnalités à de nouvelles primitives. Un nouveau prototype SDDS-2005 a été implémenté. Ses nouvelles possibilités de fonctionnement permettent la sauvegarde de fichiers SDDS sur disque, une mise à jours plus efficace et concurrentielle, l'encodage de données ainsi que la recherche par le contenu, parallèle et distribuée au sein des enregistrements

1.2 Contribution de notre Thèse

Nous avons implémenté le prototype SDDS-2005 et les différents modules le constituant. Diverses techniques ont été utilisées. Elles sont basées tout particulièrement sur une nouvelle forme de signatures, les signatures algébriques dont le calcul s'appuie sur les structures de Corps de Galois [M02, M04].

Un module de sauvegarde distribué et parallèle a été développé. Il met le contenu du fichier SDDS dans chaque RAM, sur les disques locaux des sites serveurs. Cela est nécessaire vu la volatilité de la mémoire vive. Un fichier peut aussi devenir peu utilisé occupant de la mémoire inutilement. Dans ce contexte, les signatures algébriques nous permettent de ne sauvegarder que les parties ayant été modifiées depuis la dernière sauvegarde. Un gain de temps, pouvant être très important, est constaté par rapport à la méthode naïve.

En ce qui concerne la mise à jour d'un enregistrement, l'utilisation de nos signatures permet de n'envoyer ce dernier à un serveur, pour un stockage, que si l'application l'a réellement modifié. Par ailleurs, les signatures nous permettent de garantir la sérialisibilité de ces mises à jour dans un environnement concurrent. Dans les deux cas, l'outil se révèle fort efficace.

L'ouverture des systèmes d'information des entreprises à l'Internet ne peut se faire sans le déploiement de solutions de sécurité appropriées et à grande échelle. De nouveaux services et protocoles permettent un acheminement sécurisé et sûr des données entre partenaires (IPSec, IP Security Protocols - RFC2401). L'usage d'infrastructures à clés publiques peut résoudre les problèmes de facteurs d'échelle. Ce n'est pas le but dans les SDDS. Cependant, on doit garantir au moins une protection contre les vues accidentelles par les administrateurs des serveurs notamment dans l'environnement P2P. Dans ce contexte, nous avons mis en œuvre une technique particulièrement efficace basée sur une technique dite de signatures cumulatives. Un message encodé ainsi, intercepté sur le réseau, est pratiquement

impossible à exploiter. Cet encodage est fait au niveau des clients. Il est complètement transparent aux serveurs.

De plus, cet encodage permet différents types de recherche par le contenu encodé. Ces recherches se révèlent très rapides, souvent plus rapides que par d'autres algorithmes connus tels que les algorithmes de Karp-Rabin ou de Boyer-Moore. On cite la recherche par le préfixe ayant une complexité proche de *O (1)*, la plus rapide possible dans le domaine. Nous nous sommes aussi intéressé à différents algorithmes de recherche de texte existants. Nous avons donc essayé de profiter des avantages de chacune tout en contournant leurs inconvénients, si possible. Cela est fait dans le but de déterminer la stratégie à suivre pour l'implémentation des différents algorithmes dans notre système. D'autres types de recherche sont implémentés dans SDDS-2005 notamment, la recherche partielle d'une chaîne de caractère, du plus grand préfixe ou encore de la plus grande chaîne commune.

Les techniques que nous avons mis en oeuvre sous SDDS-2005 sont nouvelles et sophistiquées. Dès lors, les résultats théoriques et l'efficacité de l'ensemble avaient besoin d'être analysés, notamment par des mesures expérimentales de performances. Nous les avons fait systématiquement. Les résultats ont confirmés nos attentes théoriques. Aujourd'hui, le prototype SDDS-2005 est parmi les prototypes les plus connus de Gestionnaire de Données Distribuées et Scalable. Enfin, il y a actuellement (Juin 2006) plus de 20000 références aux travaux sur les SDDS sur Google. Les signatures algébriques semblent d'un intérêt important dans ce concept, puisque l'article [LS04] y occupe la première place. SDDS-2005 est aussi disponible en téléchargement sur le site CERIA.

1.3 Plan de la Thèse

Le Chapitre 2 discute l'état de l'art des systèmes de gestion de fichiers distribués ainsi que l'architecture des SDDS et les protocoles de communication. Le chapitre 3 est consacré à la présentation de notre prototype SDDS-2005 et les différentes primitives ainsi que quelques aspects de programmation. On décrit par la suite, dans le chapitre 4, les signatures algébriques et les propriétés de structure de corps de Galois ainsi que l'utilisation de ces signatures algébriques dans notre prototype. On présente brièvement les signatures cryptographiques SHA-1 dans un but de comparaison avec nos signatures. Le chapitre 5 traite la sauvegarde de données en utilisant les signatures algébriques. Dans le chapitre 6, on présente une mise à jour plus efficace de données en utilisant ces signatures algébriques dans notre prototype SDDS-2005. Dans le chapitre 7, on étudie l'encodage de données dans SDDS-2005. Dans ce contexte, nous présentons les signatures cumulatives et leurs rôles pour la protection contre les vues accidentelles de données ainsi que, potentiellement, pour la protection

contre la corruption de données. Nous examinons leur utilisation dans SDDS-2005 pour les différents types de recherche par le contenu sans décodage de données. On étudie notamment la recherche par préfixe, la recherche d'une chaîne par la méthode *n*-Gramme ainsi que la recherche du plus grand préfixe et la plus chaîne commune. Nous présentons également d'autres algorithmes pour la recherche de caractères (String Matching) tels que la recherche par l'algorithme Karp-Rabin. Ensuite, dans le chapitre 8, on présente les mesures de performances relatives aux principales fonctions implémentées. Nous comparons nos résultats à quelques autres algorithmes existants. Finalement, nous concluons par un bilan de notre travail et les perspectives ouvertes par ce dernier. Nous traitons notamment les perspectives de la compression différentielle de mises à jours en utilisant les signatures cumulatives. En annexes, nous présentons l'interface offerte aux applications par SDDS-2005 ainsi que les procédures d'installation et de configuration de SDDS-2005. On trouve également des articles relatifs à nos travaux publiés dans des conférences internationales.

Chapitre

2 Les Structures de Données Distribuées et Scalables

Le besoin croissant en ressources de stockage a incité le développement de systèmes distribué à usage intensif de données aux dépens des systèmes centralisés. Notamment, l'interconnexion de plusieurs ordinateurs permet d'obtenir une puissance de calcul difficilement atteignable sinon non atteignable par les plus puissantes machines disponibles actuellement. Un système distribué constitué d'une collection d'ordinateurs connectés via un réseau sans partage de mémoire centrale est ainsi une configuration générique de plus en plus populaire. On l'appelle multi-ordinateur réseau [T95, OV99], ou Réseau de Stations de Travail « Network of Workstation » puis, plus récemment, grille de données ou Pair à Pair (Peer to Peer). Nous reviendrons sur cette configuration plus tard, étant donné qu'il s'agit de celle pour laquelle notre prototype est conçu.

Un système distribué doit supporter quelques caractéristiques telles que :

Absence de contrôle centralisé : Des clients accèdent à travers un réseau à des serveurs de fichiers sans passer par un serveur unique assurant ainsi le contrôle d'accès centralisé.

Partage de ressources : Le système doit gérer l'accès aux ressources (ressources critiques).

Concurrence : Plusieurs clients sont servis en même temps.

Tolérance aux pannes : Un système distribué doit offrir une grande fiabilité en offrant des possibilités de disponibilité permanente des données en les répliquant sur plusieurs sites.

Transparence des accès : L'accès aux fichiers distant doit être similaire aux fichiers locaux.

Les systèmes distribués s'appuient sur les architectures matérielles spécifiques. Pour la gestion de données, on utilise les systèmes de fichiers distribués. Ce chapitre survole d'abord l'état de l'art du domaine. Puis, on se concentre spécifiquement sur les structures de données scalables et distribuées.

(SDDS). On évoque par la suite les avantages de ces systèmes en présentant la notion de scalabilité. On s'intéresse aux différents types de SDDSs. Enfin, on examine les principales caractéristiques notamment l'éclatement d'une case et la migration de certaines données vers un autre serveur [D01].

2.1 Avantages des Systèmes Distribués

Un ensemble de microprocesseurs donne non seulement un meilleur rapport coût/performances qu'un gros ordinateur, mais fournit une puissance de calcul qu'aucun gros ordinateur ne pourra atteindre. Par exemple, avec la technologie classique, il est possible de construire un système comportant 1000 UC de 20 MIPS (millions d'instructions par seconde) chacune, soit une performance globale de 20 000 MIPS.

La fiabilité est un autre avantage non négligeable des systèmes distribués. La charge étant distribuée sur plusieurs machines, la panne d'un composant n'affecte qu'une machine et laisse les autres en état de fonctionnement. Nous reviendrons sur ce critère dans la section traitant la haute disponibilité.

Un autre atout des systèmes distribués, ils permettent une croissance progressive. Une entreprise achète souvent un gros ordinateur avec l'idée de lui faire exécuter tous ses travaux. Si l'entreprise prospère et que la charge de travail augmente, le gros ordinateur ne répond plus à l'accroissement en charge. Ainsi, il faudrait soit le remplacer par un autre plus puissant (s'il en existe un), soit acheter un second. Dans les deux alternatives, la vie de l'entreprise peut être perturbée. A l'inverse, les systèmes distribués permettent un développement graduel selon les besoins. L'examen du Tableau 2- 1 [G93] montre l'évolution, de 1990 jusqu'à l'an 2000, de la performance du hardware. Ces résultats encouragent de plus en plus la conception d'applications pour les multi ordinateurs.

Année	1 Chip Vitesse CPU	1 Chip DRAM	1GB Disque	Tape	LAN	WAN
1990	10 mips	4 Mb	8"	.3 GB	10 mbps Ethernet	64kbps ISDN
1995	100 mips	16+ Mb	3"	10. GB	150 mbps ATM	1mb/s T3
2000	1000 mips	64+ Mb	1"	100. GB	850 mbps ATM	1 gbps fiber

Tableau 2- 1 : Evolution de la performance du Hardware

2.2 La Communication dans les Systèmes Distribués

Les informations échangées entre différents sites de données peuvent être de nature très différentes et sous divers formats (fichiers textes, images, messages, etc...). Ces informations peuvent être situées sur

des postes distants ou encore sur des réseaux différents. Dans ce contexte de manque de standard, se pose de façon aiguë les problèmes d'intégration de logiciels divers et de conception d'applications portables. L'idée de base d'un *middleware*, la solution pour ces problèmes, est de faire abstraction des supports matériels et logiciels utilisés, et de se limiter à un mode de coopération unifiée entre applications portables. C'est une couche logicielle entre l'application cliente et les serveurs, permettant d'unifier l'accès à des machines hétérogènes indépendamment du langage de programmation des applications (Figure 2- 1a).

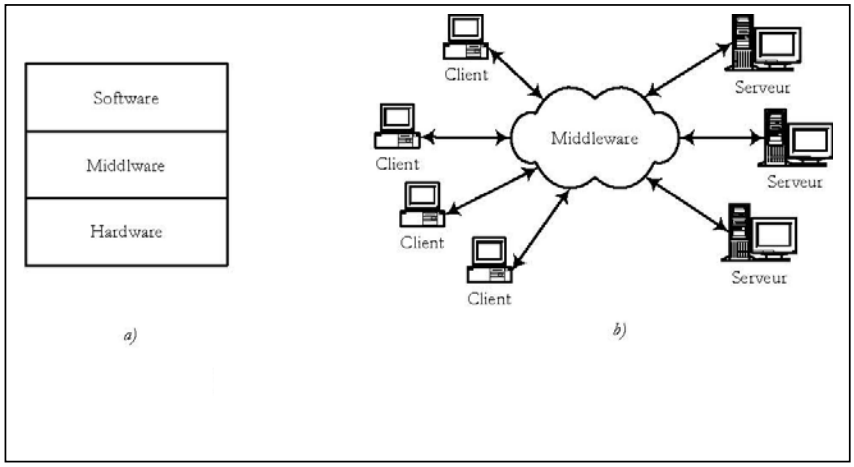


Figure 2- 1: La communication dans les systèmes distribués.

Un middleware englobe les API (*Application Programming Interface*) du côté client qui servent à demander un service, et couvre la transmission sur le réseau de la requête et de son résultat (Figure 2- 1b). Les technologies les plus marquantes dans ce domaine sont les implémentations des middlewares CORBA (*Common Object Request Broker Architecture*) [OMG98] définie par l'OMG (*Object Management Group*), DCOM (*Distributed Component Object Model*) de Microsoft [M98] et l'infrastructure EJB (*Enterprise Java Beans*) [SM98]. Le traitement distribué orienté objet représente un paradigme qui permet aux objets d'être distribués dans un réseau hétérogène et d'assurer dynamiquement les rôles de clients et de serveurs dans toute interaction.

2.3 Architectures Matérielles

Différentes architectures matérielles sont utilisées pour l'implantation des systèmes de gestion de fichiers distribués. Nous citons les plus répandues.

2.3.1 Architecture à Mémoire Partagée (Shared-Memory)

Ce sont des machines parallèles qui s'appuient sur une mémoire globale physiquement partagée par l'ensemble des processeurs (Figure 2-2).

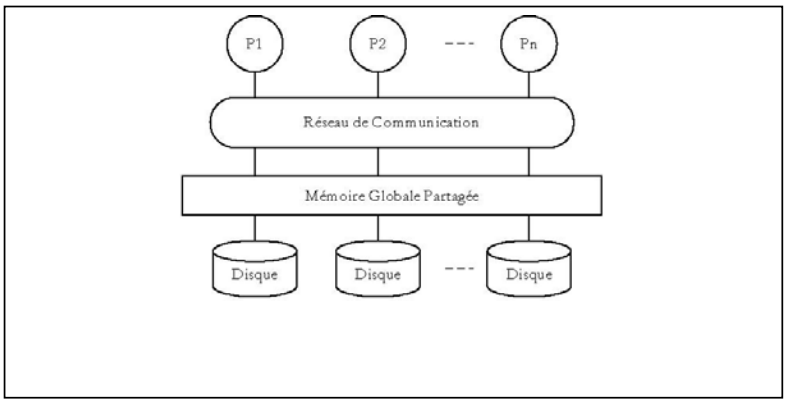


Figure 2- 2: Architecture à mémoire partagée (shared-Memory Architecture)

Cette architecture présente l'intérêt de fournir un modèle de programmation simple dans la mesure où chaque processeur accède de manière transparente à la mémoire commune. Cette approche est attrayante pour un parallélisme modéré (une dizaine de processeurs). Elle est moins utilisable pour un parallélisme massif. La mémoire devient un goulot d'étranglement lorsque le nombre de processeurs devient important. De plus, l'ajout d'un nouveau processeur peut entraîner des modifications importantes au niveau du matériel.

2.3.2 Architecture à Disque Partagé (Shared-Disk)

Dans l'approche à partage de disques, chaque processeur a sa mémoire privée et peut accéder à n'importe quel disque à travers un réseau d'interconnexion. Chaque processeur peut ainsi copier des pages d'une base de données sur un disque partagé dans son propre cache disque. La mise en œuvre d'une telle procédure nécessite une gestion des conflits d'accès aux même pages et l'implantation d'un protocole de gestion de cohérence des caches [ÖV99]. La complexité de ces mécanismes limite les performances de cette architecture.

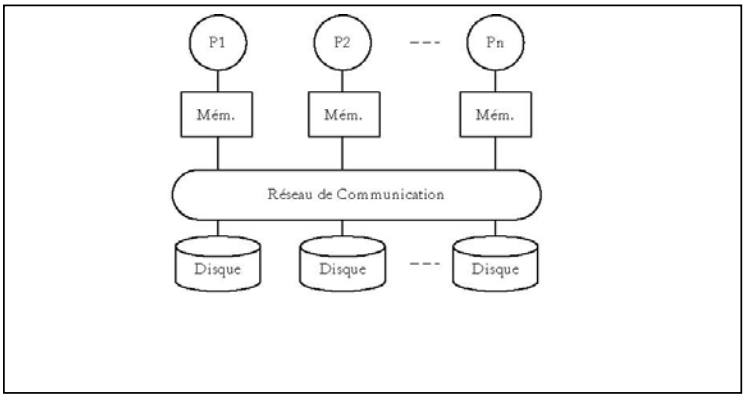


Figure 2- 3 : Architecture à partage de disque

2.3.3 Architecture sans Partage (Shared-Nothing)

Dans cette approche, chaque processeur dispose de sa propre mémoire locale et de son propre espace disque (Figure 2- 4). Chaque nœud apparaît comme un serveur de données local avec son propre base de données. Cette architecture s'est développée grâce aux réseaux de PCs et de stations de travail qui permettent d'intégrer un grand nombre de processeurs (plusieurs centaines, voire des milliers de processeurs).

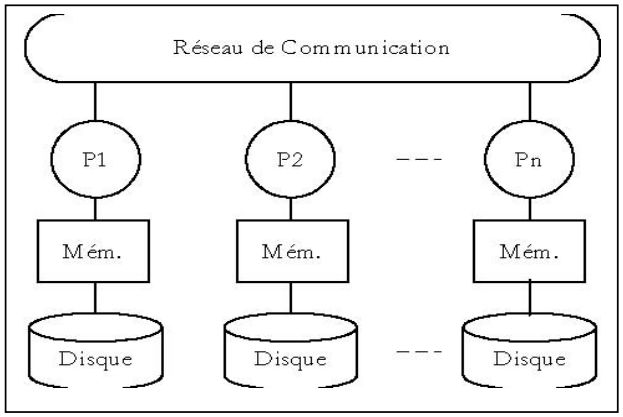


Figure 2- 4 : Architecture à partage de rien

Ce modèle est aujourd'hui très apprécié car il présente des caractéristiques d'extensibilité et de disponibilité très séduisantes. De plus, il permet de minimiser la quantité de données transférées sur le

réseau. Ces systèmes sont cependant difficiles à administrer et à programmer. Parmi ces systèmes, on cite : Peer To Peer et Grid Computing.

2.3.3.1. Architectures Pair à Pair

La technologie Pair à Pair (ang. Peer To Peer) notée aussi P2P, [AD01, S&al00], désigne un type d'architecture pour laquelle toutes les machines sont d'une même importance, en d'autres termes, à la fois clients et serveurs.

On ne retrouve plus les problèmes de goulot d'étranglement et d'extensibilité existants dans les systèmes précédents. Le rajout de ressources ne nécessite pas de grandes modifications au niveau physique.

2.3.3.2. Multi-Ordinateurs

L'augmentation régulière de la vitesse du CPU *(Central Processor Unit)*, *60%* par an, et l'optimisation des architectures ont entraîné une baisse continue du temps d'accès à la RAM *(Random Access Memory)* soit *10%* par an. Les disques durs, par contre, limités par leurs structures mécaniques, n'ont pas obtenu le même succès.

De même, les réseaux ont connu des améliorations importantes au cours des dernières années. Des recherches menées depuis plusieurs années étudient l'influence des réseaux à très haut débit sur la gestion de l'information. Les résultats montrent qu'il est actuellement plus rapide d'aller chercher une information dans la mémoire centrale d'une machine distance que sur un disque local

Ressource	Disque local	RAM distant (Ethernet)	RAM distant (réseau gigabit)	RAM local
Temps d'accès	*10 msec*	*100 µsec*	*1µsec*	*100 nsec*

Tableau 2- 2 : Temps d'accès aux données

Parallèlement, un nombre croissant d'applications exige une capacité de stockage et de traitement qui dépasse celle des plus puissantes machines actuelles. Pour répondre à de tels besoins, de plus en plus grand, une approche suivie depuis quelques années vise à combiner la puissance de traitement et de stockage d'un grand nombre de machines (tableau 2-1).

Un *Multi-ordinateur* est un réseau de stations de travail et de PCs faiblement couplés, interconnectés par un réseau d'au moins 10Mb/s (Ethernet, ATM, Fast et Giga Ethernet). Chacune de ces CPU a un accès direct à sa propre mémoire locale [G93].

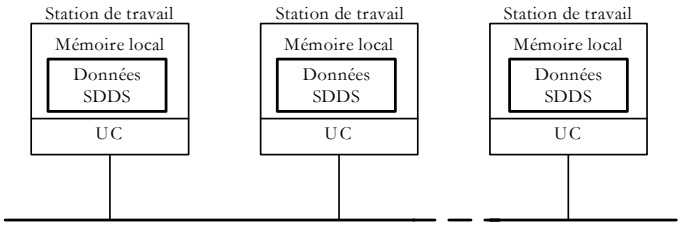


Figure 2- 5: Multi-Ordinateur

De plus, une panne au niveau d'une machine n'affectera pas le fonctionnement de la tâche du fait de la distribution de celle ci sur plusieurs machines. Une entreprise n'aura pas à remplacer une machine (gros ordinateur) par une autre plus puissante mais plutôt augmenter le nombre de ressources à travers son réseau assurant ainsi une croissance progressive. La Figure 2- 6 montre l'évolution rapide des performances de la RAM et du CPU durant les dernières 20 années.

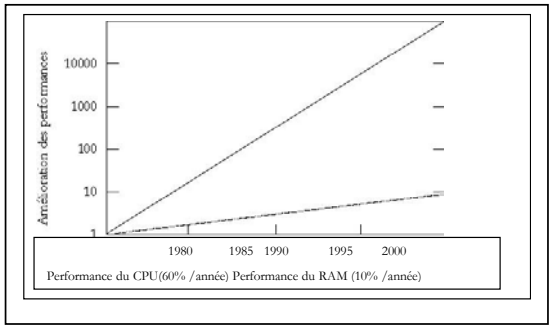


Figure 2- 6: Evolution des performances de la RAM et du CPU

2.3.3.3. Les Grilles

Une grille d'ordinateurs (ang. grid) est une infrastructure virtuelle constituée d'un ensemble coordonné de ressources informatiques potentiellement partagées, distribuées, hétérogènes et sans administration centralisée.

Le concept de *Grid* [F01] est une approche similaire au multi-ordinateur étudié plus haut. Elle a pour but de relier entre eux différents ordinateurs relativement puissants, éventuellement localisés sur des sites éloignés, afin de les utiliser, d'une part comme un seul ordinateur super puissant et d'autre part,

comme moyen de communication sophistiqué entre les différents experts intervenant dans la résolution de problèmes complexes.

2.4 La Scalabilité

La scalabilité ou montée à l'échelle d'un système [G93, G99], se résume en la capacité de supporter une montée en charge concernant les besoins en ressources. En effet, le but de tout système est d'offrir plus de rapidité de calcul sur des opérations de plus en plus complexes. Cela nécessite des machines spécialisées comportant des ressources non atteignables. La scalabilité constitue cette capacité de supporter cette extensibilité.

Un système idéal doit assurer deux propriétés clé : Le spped-up et le scale-up linéaires (Figure 2- 7).

−*Speed-up linéaire*

Les performances augmentent linéairement pour une taille de base de données (BD) constante et une augmentation proportionnelle des capacités de la configuration. Autrement dit, lorsque ces capacités sont multipliées par x ($x > 0$), le temps de réponse diminue d'un facteur de x.

−*Scale-up linéaire*

Les performances restent constantes pour une augmentation proportionnelle de la taille de la BD et des capacités de la configuration. Ainsi, lorsque la taille de la BD augmente d'un facteur de x ($x > 0$), le temps de réponse peut-être maintenu constant en multipliant les capacités de la configuration par x.

Figure 2- 7 : Le Speed-up et le Scale-up.

Le *speed-up* et le *scale-up* linéaires sont rarement atteints en pratique, car ils exigent que tous les aspects du système soient parfaitement scalables. Ce système est composé de la configuration matérielle, du système d'exploitation, de la base de données, du logiciel réseau et des applications.

On étudie la capacité de notre système à supporter divers besoins tels que le besoin croissant en espace de stockage, la variation du temps de traitement (calcul des signatures…) des structures de données.

2.5 Architectures de Systèmes de Fichiers Distribués

On décrit quelques architectures de systèmes de fichiers distribués. On présente les systèmes qui offrent l'accès transparent à un fichier distant, en général plat. Le fichier peut être, de plus, répliqué avec une copie stockée dans le cache local. On présente ensuite les architectures à fichier partitionné sur plusieurs sites. On discute différentes méthodes de partition notamment statiques et dynamiques.

2.5.1 Systèmes NFS, AFS, Frangipani et CODA

Dans ces systèmes, l'accès à des fichiers doit être transparent et leur mise à jour nécessite leur redistribution en plus de la suppression de l'ancienne version. Cette architecture s'appuie sur l'idée du stockage de chaque fichier sur un site unique. Ainsi, dans la plupart des systèmes de fichiers distribués traditionnels, chaque fichier réside entièrement au niveau d'un site spécifique alloué initialement. On retrouve plusieurs produits commercialisés tels que NFS (Network File System) [SGK+85] ou AFS (Andrew File System) [H&al88, CDK94, T92].

2.5.1.1. NFS

Dans NFS, l'idée de base est de pouvoir partager un même système de fichiers entre plusieurs clients et serveurs hétérogènes. Pour cela, un serveur NFS exporte ses *directories* pour qu'ils soient accessibles à des clients. La plus grande particularité de NFS est d'être sans état, c'est à dire qu'un serveur ne mémorise pas la liste de ses clients et donc aucune information sur leurs activités. Ainsi, puisque les caches des clients ne peuvent pas coopérer directement entre eux, les modifications distantes ne sont pas perçues comme des modifications locales. Cet effet exige que les clients accèdent fréquemment aux serveurs pour maintenir la cohérence de leur cache. NFS a été conçu pour une utilisation sur un réseau local, mais il peut aussi être exécuté au travers d'un réseau longue distance.

2.5.1.2. AFS

AFS a été développé à l'université de Carnegie Mellon et est utilisé dans l'environnement distribué Andrew. Son objectif principal est de permettre le partage de fichiers sur des domaines très étendus. Contrairement à NFS, il dispose d'un nommage global. Chaque répertoire ou fichier est identifié par un nom de chemin indépendant de la machine sur laquelle l'utilisateur est connecté. L'inconvénient majeur d'AFS est que la probabilité de défaillance d'un serveur ou du réseau croit avec sa taille. Cela génère une grande vulnérabilité aux pannes.

2.5.1.3. Frangipani

C'est un système de fichiers distribués développé dans l'environnement Unix 4.0 [TML97]. Il est bati sur un système de disques virtuels distribués appelé Peatl [LT96], implémenté de manière décentralisée. Il pêrmet le partage de fichiers sur un ensemble de disques gérés comme une seule unité de stockage. Il est plutot destiné aux machines en grappes ou cluster (ang. Cluster machines) avec une administration commune et supportant un système de communication protégé. Il fournit une vue glabale et cohérente du système, en utilisant un protocole efficace de mise à jour de l'état du système ce qui permet d'éviter les accés inutiles aux serveurs.

2.5.1.4. CODA

CODA est une évolution d'AFS, il reprend beaucoup de ses principes tels que la gestion par volumes et l'utilisation de la réplication. Il ajoute la gestion des opérations en mode déconnecté. C'est à dire qu'un client peut rapatrier sur son poste les fichiers dont il a besoin, se déconnecter du réseau et continuer à travailler normalement sur ses fichiers. Quand la connexion sera rétablie, Coda se charge de mettre à jour les fichiers, en gérant aux mieux (ou en demandant à l'utilisateur) les éventuels conflits qui peuvent apparaître. L'écriture retardée est aussi une nouveauté de Coda par rapport à AFS, cela permet de retarder l'écriture d'une modification sur un fichier afin de pouvoir les regrouper et ainsi optimiser les performances [C88].

Pour gérer les cas "déconnectés", lorsqu'un utilisateur accède à un fichier, il ne travaille pas directement sur le contenu du fichier stocké sur l'un des serveurs, mais sur une copie locale du fichier qui a été mise dans un cache (une partition dédiée ou un fichier) lors du premier accès. L'architecture de Coda a été simplifiée par rapport à celle d'AFS, en plus du module noyau interceptant les appels systèmes sur les fichiers partagés, il y a uniquement deux serveurs. "Venus", sur le client, qui sert de gestionnaire de cache et Vice, sur le serveur, qui prend en charge la gestion des fichiers.

2.5.2 Architecture à Partition d'un Fichier

Dans cette architecture, on retrouve plusieurs méthodes de partitions pour un fichier. Les principales sont ; le « Round-Robin » [Ter88], la partition par hachage, la partition par intervalle [D&All86] et la partition multidimensionnelle. Nous présentons d'abord les principes généraux correspondants. Ils ont été utilisés historiquement pour créer les fichiers partitionnés statiquement, ou sur un nombre limité de

sites. Ensuite nous abordons spécifiquement la partition dynamique caractéristique des structures de données distribuées et scalables.

2.5.2.1. Round Robin :

C'est le schéma de partition le plus simple. Elle permet de partitionner un fichier sur les nœuds de stockage en fragments de même taille. Lorsqu'un site doit répondre à un grand nombre de requêtes, il doit utiliser une technique de répartion de charge (load balancing) entre plusieurs serveurs. 'Round Robin' est une technique qui se place directement au sein du serveur DNS et qui consiste à attribuer successivement les différents serveurs à la manière d'un tourniquet. Appliquée à une relation, elle repartie les tuples successives d'une manière similaire. Elle garantit la distribution la plus équilibrée des données. *Round Robin* est une excellente méthode pour des requêtes parallèles qui demandent tous les tuples d'une relation.

D'autres systèmes permettent le partage de fichiers sur plusieurs sites différents. Il s'agit de fichiers structurés ou plats. On retrouve plusieurs méthodes de partition:

2.5.2.2. Partition par Hachage (Hashed Partitioning)

Cette méthode consiste à l'application d'une fonction de hachage *h* à l'ensemble des valeurs de l'attribut du partitionnement. Si *a* est une valeur de l'attribut du partitionnement, alors *h* (*a*) retourne une valeur permettant de l'assigner à un nœud. En général l'ensemble des valeurs possibles de la fonction de hachage est subdivisé en *n* intervalles, où *n* représente le nombre de nœuds possibles pour cette partition.

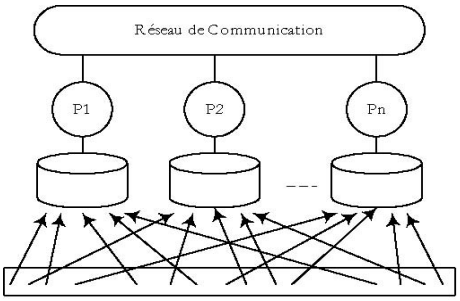


Figure 2- 8: Hached partitionning

2.5.2.3. Partition par Intervalle (Range Partitioning)

Cette approche permet de partitionner l'ensemble des enregistrements en se basant sur les valeurs de clés de ces enregistrements (Figure 2- 9). La méthode consiste en deux phases : (*1*) il s'agit d'abord de

diviser l'espace des valeurs des attributs sélectionnés en intervalles puis chaque intervalle est affecté à un nœud, (*2*) ensuite, chaque enregistrement est assigné au nœud *n* si la valeur de sa clé appartient à l'intervalle de *n*.

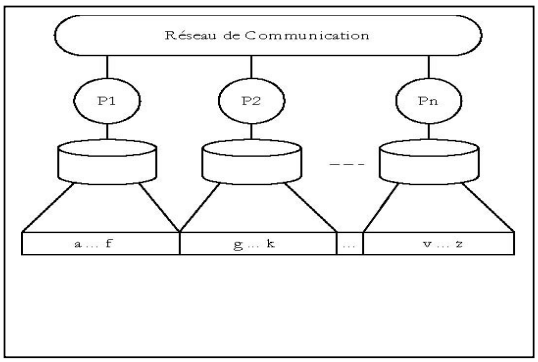


Figure 2- 9: Range Partitionning

Cette méthode permet de résoudre un certain nombre de problèmes associés l'algorithme *Round-Robin*, en particulier dans l'exécution des 'requêtes à conditions' sur les attributs de partition.
Cependant, ce type de partition ne garantit pas que les fragments issus aient la même taille. De ce fait, certains fragments peuvent être disproportionnés par rapport à d'autres. Ce scénario peut parfois se présenter dans son cas extrême avec des nœuds vides ou encore avec un fragment qui contient la totalité des enregistrements.

2.5.2.4. Partition Multidimensionnelle (MDS)

Les méthodes traditionnelles pour l'accès aux données multidimensionnelles (plusieurs critères) nécessitent l'emploi de tables d'indexe secondaires organisées sous forme de listes inversées. Ces méthodes se sont avérées très coûteuses en espace et en temps [VBW98].

Des méthodes modernes sont apparues entre les années 80 et 90. La plupart de ces méthodes sont à bases de hachage dit 'hachage dynamique'. Dans ce domaine, on citera les travaux de Nardelli pour la localisation des données utilisant le concept de tables de hachage distribuées. Ces méthodes n'utilisent pratiquement pas d'indexe et permettent un accès très rapide à l'information. Cependant, on retrouve différents problèmes dans ces systèmes tels que le chevauchement provoquant l'accès inutile à des régions lors de la recherche de données et les goulots d'étranglements. Ces structures sont alors peu scalables.

2.6 Partition Dynamique Distribuée

Suivant les schémas de partitionnement statique, un fichier ne peut être réparti que sur les sites alloués initialement. Ainsi, lorsque la distribution est établie, le nombre de sites et les critères de partitionnement sont figés, même si l'exploitation du fichier devient non optimale par suite des insertions. Il faut alors réorganiser la totalité du fichier. Cette opération peut être complexe et coûteuse, en temps notamment, pour un grand fichier. Les schémas de partitionnement dynamique ont été conçus pour surmonter cette difficulté. Le nombre de sites d'un fichier peut augmenter d'une manière incrémentale quand le fichier grandit.

La partition « *Round Robin* » ne se prête pas aisément à une version dynamique. Par contre on a généralisé dans ce sens, progressivement, les autres méthodes de partition discutées. Les généralisations ne sont pas triviales. On a conçu d'abord un certain nombre pour relativement peu de sites. Nous mentionnons une de 1ères structures de ce type ci-dessous. Puis nous discutons plus en détail les SDDSs que l'on a commencé à proposer depuis 1993, pour des extensions limitées en pratique seulement par le nombre de site disponibles pour la partition.

2.6.1 Le Hachage Linéaire Distribuée (*DLH*)

La méthode DLH (ang. Distributed Linear hashing) a été proposée pour la partition dynamique par hachage sur super-ordinateur (la machine Buterfly), [SPW90]. Un fichier est partitionné dynamiquement par le hachage linéaire. Il peut s'étendre sur un nombre limité de mémoires et de processeurs. Ses paramètres courants sont cachés dans le cache de chaque processeur. Les caches sont rafraîchis d'une manière synchrone lorsqu'il y'a un repartitionnement, par un éclatement de hachage linéaire. Il en résulte une adjonction d'un processeur avec sa mémoire au fichier.

2.6.2 Structure de Données Scalables et Distribuée (*SDDS*)

Bien que les concepts tels que 'P2P' ou 'Grid Computing' n'étant pas parfaitement identiques, ils visent tous, la conception de plates formes capable d'exploiter cette énorme masse de ressources qui d'ailleurs, apparaît largement inutilisée ou du moins, peu utilisée aujourd'hui. De nouvelles structures de données et de gestion de ces données réparties sont alors nécessaires pour tirer profit des potentialités offertes. Dans ce contexte, Les Structures de Données Distribuées et Scalables (SDDS) sont une nouvelle classe de structures proposées dans ce but, et définie spécifiquement pour la gestion de fichiers distribués sur un multi ordinateur [G93].

En quelques mots, une SDDS (ang. Scalable Distributed Data Structures) n'a pas de répertoire central et peut s'étendre d'un seul site à *un nombre indéterminé de sites*. Le fichier SDDS est structuré en enregistrements identifiés par clés. Les données peuvent résider entiérement en mémoire vive distribuée. [LNS93a] [LNS93b]. Les principes des SDDS sont destinés à des multi-ordinateurs plus spécifiquement. Néanmoins, ils s'appliquent aussi aux configurations dites aujourd'hui Pair à Pair (P2P), (ang. Peer to Peer), et celles dites Grilles de Données (ang. Grid computing) ou encore les Systèmes de gestion de fichiers (SGF). On s'intéressera plutôt à cette dernière possibilité.

Dans les systèmes traditionnels, la taille d'un fichier est limitée par l'espace disque disponible sur le site. Un fichier SDDS n'a pas de répertoire central et peut s'étendre d'un seul site à un nombre indéterminé de sites. Il fournit un mécanisme général d'accès à des données réparties dynamiquement et structurés sous forme d'enregistrements. Ces fichiers s'adaptent dynamiquement à l'accroissement du volume de données. Ces structures disposent en plus, du traitement parallèle et une capacité de stockage potentiellement illimitée. En même temps, elles offrent des temps d'accès beaucoup plus courts que les performances d'accès aux données stockées sur les disques. Ces caractéristiques assurent aux SDDS des performances de traitement supérieures à celles des structures de données traditionnelles. Nous présentons les principes des SDDS en détail plus loin.

2.7 Principes des SDDS

Les SDDS sont basées sur le modèle Client / Serveur. Il y a aussi le concept d'un *pair* dans cette approche, c'est un site qui assure les deux fonctions à la fois. Les SDDSs permettent la création de fichiers scalables répartis sur les serveurs. Ces fichiers peuvent être stockés en mémoire vive distribuée. Cette conception est avantageuse, car il est beaucoup plus rapide d'accéder à des données se trouvant dans la mémoire d'une machine distante que sur un disque, même local [G93].

L'architecture SDDS permet à un fichier d'être composé d'un nombre illimité d'enregistrements en l'étendant d'un seul site de stockage à un nombre indéterminé de sites (serveurs). Cela se fait par des *éclatements* de *cases* (ang. buckets) sur les serveurs qui contiennent les enregistrements du fichier. Chaque éclatement évacue une partie du contenu d'une case, la moitié en général, vers une case nouvelle, créée pour le fichier sur un autre serveur. A l'éclatement et dans le cas SDDS-2005, celui ci ne doit pas contenir de case pour ce fichier. Tout éclatement est déclenché par une insertion qui déborde la capacité d'une case. L'éclatement peut alors concerner cette case. C'est le cas de SDDS RP* que nous décrivons dans la Section 2.6.2 et de bien d'autres SDDSs de type LH*. Il peut s'agir alternativement d'une autre case, choisie par un site dit *coordinateur*. Une politique des enchères est appliquée, elle

consiste à envoyer une demande d'offre à un groupe de sites puis d'effectuer des évaluations sur les réponses reçues. Pour certains types de SDDS, cette technique se base sur la taille de la mémoire physique des sites cibles candidats [T95].

Les éclatements partitionnent dynamiquement le fichier. Divers types de partitionnement sont possibles. Les principaux sont : par hachage, ou ordonné, c. à d par intervalles de valeurs de clés primaires, ou enfin multidimensionnels (multi-clés). Nous discutons davantage les SDDS plus loins. Indiquons par avance, et à titre d'exemple, que dans le cas de SDDS-2005 nous utilisons des SDDS du type RP* (ang Range Partitioning) qui utilise la partition par intervalle.

Les cases sont accédées à partir des clients SDDS. Chaque client possède en général une *image* du fichier qui lui indique l'état de la partition. Cependant, les clients ne sont pas informés des éclatements d'une manière synchrone. Le serveur qui éclate sa case ne connaît aucun client d'ailleurs. Un client peut donc adresser une requête à une case *incorrecte*. Celle ci a alors la charge de faire suivre cette requête vers la case *correcte*. L'un des aspects les plus intéressants de la conception d'une SDDS, est de pouvoir accéder à un enregistrement donné en un minimum de messages de redirection (ang forwarding ou hops). Ceci, aussi bien en moyenne que dans le pire des cas. Les minimums théoriques sont à l'heure actuelle caractéristiques de la SDDS LH* que nous présentons plus bas. Indiquons seulement, qu'il s'agit de deux messages de redirection dans le pire de cas. Mieux, on a montré tout récemment que si la requête émane d'un noeud pair de LH*, alors ce nombre minimum peut être réduit à un seul message seulement.

Un autre aspect d'une conception d'une SDDS est l'*ajustement* d'une image. Il s'agit d'une évolution de celle-ci, à la suite de celle du fichier. L'évolution de l'image est due aux messages dits IAM (ang. Image Adjustment Message). Une IAM est en général expédié par le serveur de la case correcte ayant reçu un message redirigé. Les détails de ce protocole et de l'algorithme d'ajustement dépendent de chaque SDDS. Une mise à jour intervient uniquement lorsque le client commet une erreur d'adressage.

2.8 Typologie des SDDS

Il y a plusieurs critères de typologie pour les SDDS. La Figure 2- 10 montre certaines, ainsi que les principales SDDSs proposées, classées en conséquence. La typologie primaire est celle du type de partition générée. On distingue alors les familles suivantes : les SDDS basées sur le hachage, celles qui sont ordonnées et celles basés sur les données multidimensionnelles D'autre critères concerne la haute

disponibilité, l'accès multi clés ou encore la sécurité de données. Nous détaillons ces critères dans le chapitre 3.

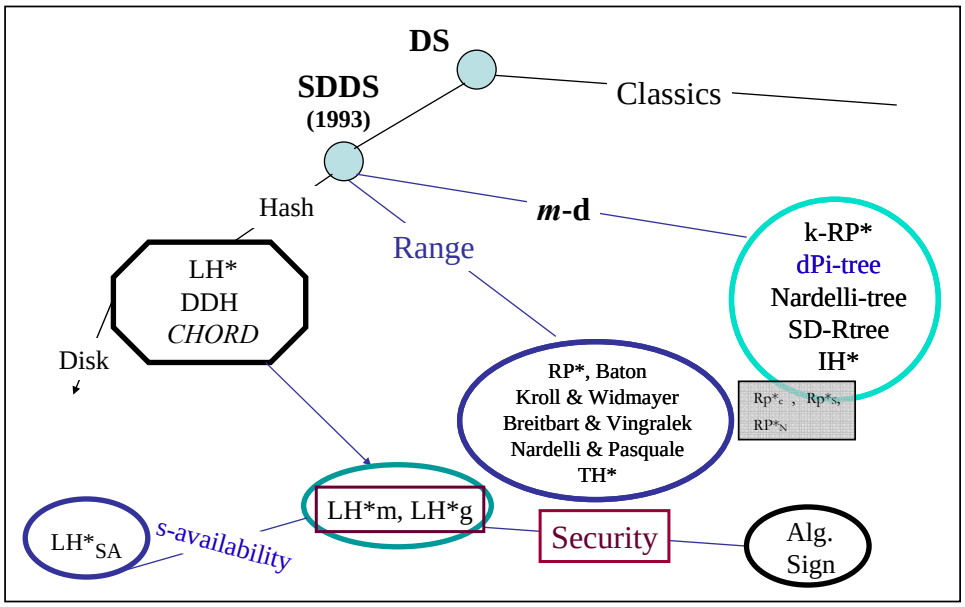


Figure 2- 10*: Les SDDS connues*

2.8.1 Partition par Hachage

Plusieurs SDDSs ont utilisé le hachage pour partitionner les données distribuées. Comme pour les structures centralisées, il s'est avéré que le hachage est particulièrement efficace pour l'adressage scalable et distribué à partir de la clé (il est naturellement moins pour d'autres types de requêtes, notamment à intervalle). La première SDDS publiée avait été ainsi basée sur le hachage linéaire, il s'agit de le LH* [LNS93]. LH [L80] est un algorithme de hachage extensible par lequel on étend l'espace d'adressage primaire d'un fichier disque pour éviter l'accumulation de débordements et la détérioration des performances d'accès.

LH* a donné lieu à plusieurs variantes. Citons d'abord le LH*$_{LH}$ [KLR94] qui vise l'architecture multiprocesseur à partage de rien. Il y a ensuite les versions de LH* à haute disponibilité [LS00] : LH*$_{RS}$ Elles permettent notamment la reconstitution de données en cas de panne d'un ou plusieurs serveurs. On utilise alors des données de parités stockées sur les sites dits de parité utilisant les codes de Reed Solomon [RS60] permettant notamment la reconstitution de données. D'autres SDDS se basent sur LH*. On cite LH*g qui implante le concept de groupage. Chaque groupe est supplémenté d'un serveur

de parité, calculé par 'XOR' des serveurs de données, et permettant au groupe de survivre à l'échec d'un serveur [LR97, L97]. On cite également les travaux de Devine [D93], Karlsson [K98], et Bennour [B00a]. Ce dernier travail était la 1ère implémentation de LH* sous Windows.

D'autres SDDS basées sur le hachage ont été également proposées. Citons d'abord le Distributed Dynamique Hashing (DDH), [D93]. Ce dernier est une version scalable et distribuée du hachage dynamique DH. Ce dernier a introduit notamment le concept, fort populaire aujourd'hui, d'une Table de Hachage Distribuée (ang. Distributed Hash Table, DHT). Les systèmes DHT tentent de résoudre le principal problème de systèmes 'Pair à Pair' décentralisés (Gnutella, …) [AH00, GN01]. Elles tentent aussi de minimiser le nombre de messages transmis en cherchant à localiser efficacement le ou les destinataires d'un message alors que les anciens systèmes utilisent souvent des mécanismes de routage simplistes à l'aide d'un broadcast massif des messages. Le principe est d'attribuer à chaque « case » de la table, un index obtenu via le hachage d'une « clé » qui est souvent une chaîne de caractères. En fournissant la clé 'haché', on peut alors lire l'élément correspondant. Puis, les travaux de Vingralek & al, [VBW94], proposent un schéma de distribution de données où plusieurs cases d'une SDDS peuvent cohabiter sur un même nœud avec un contrôle local de charge pour chaque nœud.

Enfin, récemment, il y a le CTH* (ang. scalable and distributed Compact Trie Hashing), avec sa généralisation scalable et distribuée du hachage digital (ang. Trie Hashing) [Z04]. Le CTH* génère un arbre digital scalable et distribué, dérivé de l'arbre spécifique (ang. trie) du hachage digital. Celui-ci partitionne le fichier selon un ordre total des clé. Le CTH* représente son arbre sous la forme compacte, en pré ordre, sans pointeurs internes. Cette organisation vise surtout le traitement efficace de requêtes à intervalle (comme le hachage digital en général).

2.8.2 Partition par Intervalle

Plusieurs SDDS ont été proposées pour ce type de partition [LNS94]. Les schémas RP* ont été les plus étudiés et nous concentrons notre attention sur cette approche. Ainsi, les enregistrements dans un fichier RP* sont stockées par ordre de leurs valeurs de clés de telle sorte qu'une clé et son successeur se suivent dans la même case. Chaque fragment d'un fichier est stocké dans une case pouvant contenir un maximum de *b* enregistrements. Ces fragments sont stockés au niveau de la mémoire centrale des serveurs. Chacune des cases, lui est associée un intervalle de clés borné par une clé minimale et une clé maximale. Ainsi, un enregistrement de clé *c* appartiendra à la case d'intervalle $[C_{Min}, C_{Max}]$ telle que :

C_{Min} <c≤ C_{Max}. L'union des intervalles partitionne l'espace des valeurs des clés. Les cases peuvent être indexées sur le client ou les serveurs. Une case peut être structurée en interne en un *arbre B+* spécifique pour la mémoire centrale [D01].

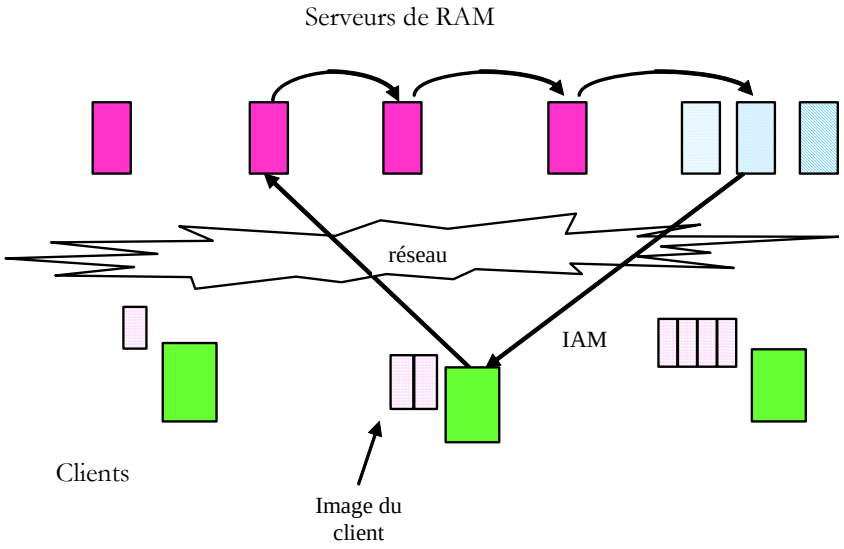


Figure 2- 11 *:* Evolution de l'image de client accédant au fichier

Initialement, un fichier RP* est composé d'une seule case avec l'intervalle initial. Toutes les insertions se font dans cette case jusqu'à dépassement de sa capacité maximale. Un éclatement est alors provoqué, transférant tous les enregistrements (au nombre de $b/2$) ayant de clés supérieures à la clé médiane de la case vers un nouveau serveur [LNS94].

Le mécanisme d'éclatement présenté est commun aux principaux schémas de type RP*, nommées $RP*_n$, $RP*_c$ et $RP*_s$. Un client $RP*_n$ envoie les requêtes aux serveurs en utilisant uniquement des messages multicast. Il n'a pas d'image de la répartition courante du fichier existant sur les différents serveurs contrairement à un client du type $RP*_c$ qui dispose d'une image. Celle-ci lui permet d'utiliser des messages unicast à chaque fois qu'il s'adresse à un serveur déjà référencé dans l'image. Dans le cas contraire, il utilise un message multicast adressé à un serveur déjà connu, le plus proche dans l'ordre des intervalles de celui ou ceux à trouver.

Le serveur recevant une requête à un fichier RP* vérifie d'abord si la clé appartiens à l'intervalle de sa case. Si la requête est un multicast et la clé est hors l'intervalle, alors elle est ignorée. Si c'est une requête unicast et la clé est hors intervalle, alors le serveur d'un fichier $RP*_n$ ou $RP*_C$ multicaste la requête à l'ensemble des serveurs en ajoutant son adresse et son intervalle de clés. Le serveur de type $RP*_S$ renvoie le message à un serveur d'indexe spécifique distribué. Ce serveur permet la redirection de la requête par le biais d'un message unicast. Les données de la case initialement adressée à tort et de celle correcte, sont finalement transmises au client dans le message IAM (ang. Image Adjustment Message). Elles servent alors à l'ajustement de l'image du client. La Figure 2- 12 ompare ces trois schémas.

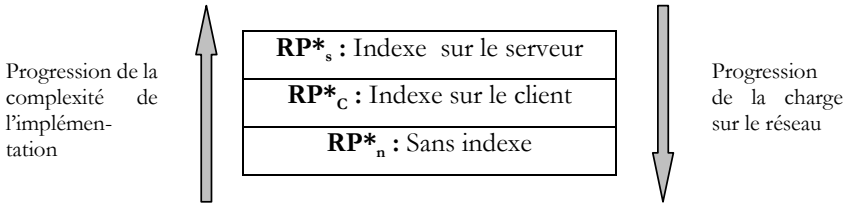


Figure 2- 12: Famille RP*

L'utilisation d'index décharge le réseau des renvois de messages. Néanmoins, cela augmente la compléxité d'implémentation des différents composants SDDS.

2.8.2.1. RP* dans SDDS-2005

Dans SDDS 2005, un même serveur peut gérer plusieurs fichiers SDDS. Chaque serveur peut ainsi avoir plusieurs cases de fichiers différents. Cette démarche a nécessité la mise en place d'un serveur de noms des fichiers, assurant la validation des noms identifiant les fichiers de manière unique. Chaque serveur utilise localement une table d'allocation des fichiers appelée *BAT* (*Bucket Allocation Table*) pour gérer ces cases.

2.8.3 Partition Multidimensionnelle

Plusieurs variantes de structures d'indexs multidimensionnelles ont été proposées. On cite: les structures M-tree, R-Tree et k-d-tree. Elles se servent de table de hachage dynamiques et distribuées pour la localisation. Ces méthodes n'utilisent pratiquement pas d'index et permettent un accès très rapide à l'information. De nouvelles structures de données distribuées et scalables (SDDS) ont vu le jour notamment VBI (Virtual Binary Index). Sa conception est inspirée de BATON (Balanced Tree Overlay Network) mais plutôt adaptée aux données multidimensionnelles.

On distingue dans VBI, deux types de nœuds: les nœuds internes dit de 'routage' et des nœuds feuilles dit de 'données'. La distribution de VBI sur un réseau P2P est à raison d'un noeud par pair. On obtient alors une version distribuée de VBI.

2.9 Requêtes à une SDDS

Dans les sections précédentes, nous avons parlé de requêtes à partir d'une clé d'un enregistrement. Plus précisément, on distingue dans une SDDS trois types de requêtes. Il s'agit d'abord de ***requêtes à clé*** (ang. *key based request*). Une telle requête recherche, insère, supprime ou encore met à jour un enregistrement dont la clé est précisée dans la requête Celle ci adresse la case correcte, ou celle concernée par la redirection.

Ensuite, on a les ***requêtes à intervalle*** (ang *range queries*). Dans ce type de requêtes, on recherche tous les enregistrements dont les clés appartiennent à un intervalle appelé 'l'intervalle de la requête'. Une telle requête peut être envoyée à l'ensemble des serveurs par un message multicast puis traitée par chaque serveur dont l'intervalle de clés correspond à celui de la requête. Les résultats sont ensuite envoyés aux clients de façon parallèle [D01]. Alternativement, une requête à intervalle peut être envoyé par unicast à certaines cases seulement, selon l'image du client. Il peut y avoir alors des redirections, qu'il faut minimiser bien sûr. Le client dispose de deux stratégies pour terminer une requête de recherche à intervalle. L'une *déterministe* où chaque serveur *S* envoie une réponse au client avec son intervalle de clés $(\lambda, \Lambda]$ et, éventuellement, les enregistrements trouvés. Le client termine la requête avec succès après comparaison de son intervalle avec $(\lambda, \Lambda]$. L'autre approche est *probabiliste.* . Le client utilise un temporisateur ou *time out t* pour collecter les réponses. Le serveur répond lorsqu'il trouve au moins un enregistrement ayant une clé dans l'intervalle de la requête. Le *time out* est réinitialisé après la réception de chaque réponse et lorsqu'il expire, la requête est terminée.

Enfin il y a les ***requêtes parallèles globales*** (ang. *scans*). Elles sont envoyées par unicast ou multicast à toutes les cases. Dans le cas unicast, il faut un protocole de *diffusion* garantissant que toute case reçoit le « scan » qu'une seule fois. Les détails dépendent de la SDDS spécifique. Voir par exemple [LNS96] pour LH*.

2.10 Synthèse

Dans ce chapitre, nous avons présenté une vue générale sur les architectures matérielles distribuées. Nous avons ensuite présenté les différentes architectures de systèmes de fichiers distribués ainsi que des techniques de placement des données pratiquées sur ces architectures. Nous avons montré que le

placement statique des données freine les performances et entraîne une vulnérabilité face aux pannes. Par contre, la répartition dynamique fournit un mode de gestion décentralisé et extensible qui induit des gains importants en terme de performance et de disponibilité. Puis, nous avons présenté une introduction aux SDDSs, définissant de nouvelles structures de données pour la gestion dynamique de grandes masses de données réparties et supportant une montée à l'échelle notamment au niveau des architectures appelées multi ordinateurs ou encore *GRID*. Les SDDSs, définies en particulier pour ces architectures, possèdent des caractéristiques permettant de prévoir des temps d'accès inaccessibles aux structures de données classiques.

Nous avons aussi décrit la famille SDDS et les différents types de partition notamment celles par hachage et par intervalle. Nous avons décrit la famille SDDS RP* sur laquelle repose notre travail ainsi que les principes généraux des SDDS et les différents types de requêtes.

Chapitre 3 *Prototype SDDS-2005*

Ce chapitre décrit notre prototype SDDS-2005. On présente d'abord son architecture générale qui repose sur l'utilisation de fichiers SDDSs RP*. Le système supporte, dans ce cadre, des cases de données de différents fichiers sur un même serveur. On présente ensuite les divers composants de notre système avec leurs caractéristiques. Puis, on discute les fonctions offertes à l'application. On centre la présentation sur les fonctions nouvelles par rapport à SDDS-2000. Il s'agit essentiellement de la sauvegarde d'un fichier SDDS sur disque, du traitement concurrentiel des mises à jour d'enregistrements, de l'encodage de données ainsi que la recherche par le contenu dans ces enregistrements. Nous donnons un bref aperçu de techniques de programmation par les threads et les sockets, utilisées lors de l'implantation de notre système. On montre les mécanismes d'utilisation de protocoles UDP ainsi que TCP/IP pour la communication sous SDDS-2005. Enfin, on aborde le contrôle de flux et la sécurité de données dans SDDS-2005.

3.1 Composants de SDDS-2005

L'architecture générale de SDDS-2005 s'articule autour de trois types de composants (Figure 3- 1) : Les serveurs de données, les clients et le serveur de noms. Par ailleurs, un site SDDS peut être *pair*, c'est à dire client et serveur en même temps.

Les Serveurs SDDS :

Un serveur SDDS scrute la réception de requêtes. Il se charge d'exécuter la tâche demandée par le client. Par la suite, une réponse est éventuellement envoyée à ces clients.

Les Clients :

Un client permet aux applications de formuler leurs requêtes sur les différents fichiers SDDS. Cela est fait d'une façon transparente par rapport à l'application. Plusieurs clients peuvent exister dans un même site.

Le Serveur de Noms

Un serveur de noms s'occupe de l'attribution de noms de fichiers et leurs unicités. Il doit être consulté par les serveurs de données concernés avant chaque création d'un fichier.

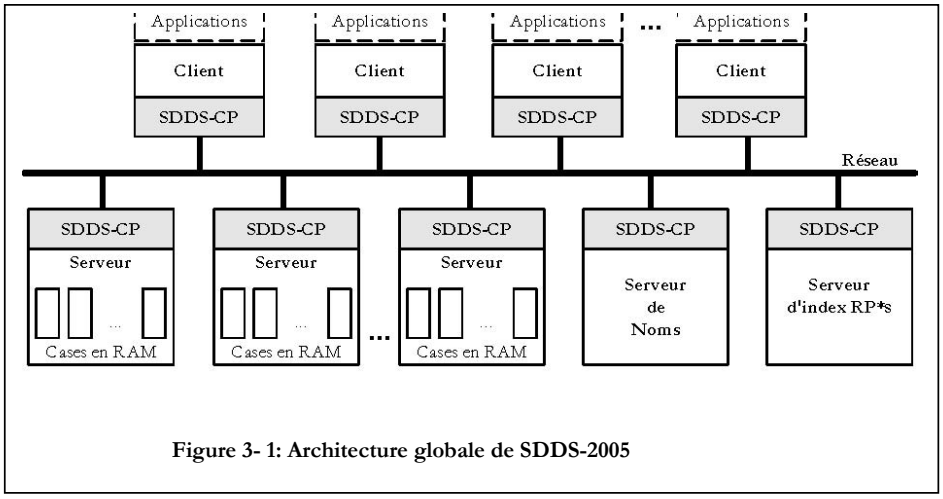


Figure 3- 1: Architecture globale de SDDS-2005

3.1.1 Serveurs SDDS

L'architecture de traitement d'un serveur SDDS 2005 (Figure 3- 2) se base sur les processus légers (ang *thread*) concurrents (ang. *multithread architecture*). Un processus léger est créé pour traiter chaque requête d'un client à travers un port qui lui est réservé. Cette architecture est plus performante que celle plus conventionnelle dite de processus *itératifs*, prenant les requêtes séquentiellement. Le nombre de threads de travail peut être changé dynamiquement en fonction de la charge du serveur.

Un serveur peut supporter plusieurs cases appartenant à des fichiers différents. La gestion interne des cases est réalisée au niveau du serveur (initialisation d'une case, stockage des données, gestion des indexs, éclatement d'une case qui déborde…etc).

Par ailleurs, le serveur se sert de mécanismes assurant son dialogue avec les autres composants de SDDS 2005. Elles concernent la réception des messages et leur décomposition, le formatage de ces messages suivant la structure attendue par les récepteurs et leurs envois au client.

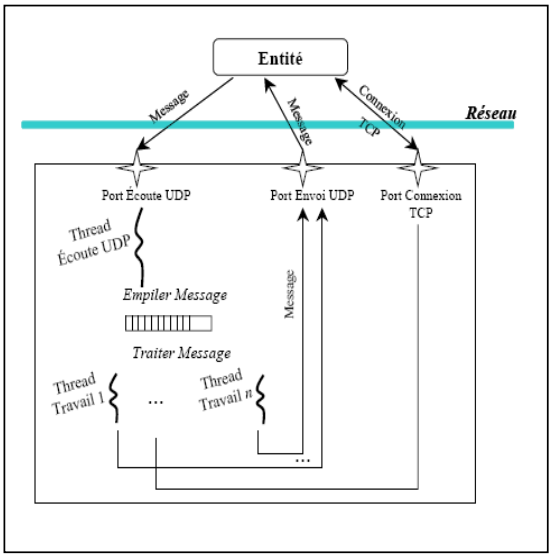


Figure 3- 2 : Architecture d'un serveur SDDS-2005.

Les différentes fonctionnalités d'une case sont assurées par deux types de *threads*, le *Thread Ecoute UDP* et des *Threads de Travail.*

Un premier *thread d'écoute,* sur un port du réseau, reste à l'affut de l'arrivée d'une requête émise par un client SDDS. Cette requête est mise dans la file d'attente des requêtes reçues. Enfin, ce thread signale la présence de requêtes à traiter aux *Threads de Travail* par l'événement *ExisteRequête.*

Le groupe de *threads de travail* attend l'arrivée d'une requête à traiter. Celle ci est identifiée puis traitée par un thread de travail qui la prélève de la file d'attente. Enfin, un traitement lui est associé

3.1.2 Client SDDS

Un client SDDS supporte tous les algorithmes de la famille RP*. Il propose à l'application locale un service d'accès aux données des fichiers SDDS. Il est vu comme un service Windows2000/NT chargé de recevoir les requêtes de l'application puis de les acheminer vers les serveurs. Il commence par structurer les requêtes en mettant en forme tous les paramètres afin de constituer le message acheminé aux serveurs. Puis, Il se charge de recevoir les réponses de ces derniers et de les transmettre aux applications respectives. Ce mécanisme masque tous les détails de l'implémentation aux applications. Celles ci n'ont pas à se soucier de la localisation des données ni des serveurs qui les gèrent.

Le client dispose de deux modules (Figure 3- 3). D'abord le *module d'envoi* des requêtes. Celui ci réceptionne les messages venant des applications locales puis les achemine aux serveurs correspondants en se basant sur son image (cas de RP*$_c$ et RP*$_s$). Une correspondance est faite entre chaque requête envoyée et l'application correspondante. Ensuite, le *module de réception* des données et des acquittements provenant des serveurs sur les ports correspondants. Il s'occupe de l'extraction de données et leur transmission à l'application correspondante dans un format convenable. Dans le cas d'une requête simple, la réponse attendue peut être accompagnée d'une image *IAM*, utile pour la mise à jour de l'image du fichier sur lequel la requête initiale a été appliquée.

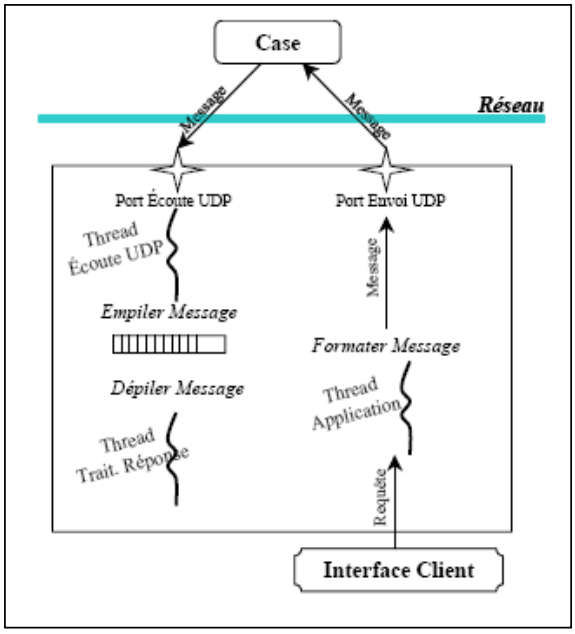


Figure 3- 3 : Architecture d'un client SDDS-2005.

Les différentes fonctionnalités d'un client sont assurées par trois *threads*. D'abord, *le Thread Application* qui affiche un menu à l'utilisateur, lui permettant d'exprimer une requête ou d'exécuter un traitement. Une liste de choix est alors proposée, permettant de formuler des manipulations élémentaires sur le fichier, on cite : l'insertion, la suppression, la mise à jour d'un enregistrement, et bien d'autres fonctionnalités telles que la recherche d'une chaîne de caractères dans les enregistrements. Ensuite, le *Thread d'écoute UDP*. Il est initié à la création d'un client et il reste à l'affût des messages entrants. Il écoute sur le *Port Ecoute UDP Client*, empile les messages reçus dans la file des réponses reçues, et signale la présence de messages à traiter au *Thread de Traitement de Réponse. Celui ci* reste à l'affût d'un

signal de l'événement *ExisteMessage*, pour dépiler un message et le traiter. Ces messages peuvent être des réponses aux requêtes, des messages d'ajustement d'image ou des messages d'information : changement d'adresse…etc.

3.1.3 Serveur de Noms

Le serveur de noms joue un rôle primordial dans le système SDDS-2005. Il assure l'attribution des noms de fichiers SDDS et leur unicité. Ces derniers sont enregistrés dans une table appelée *annuaire des noms* que le serveur de noms consulte à chaque requête de création d'un fichier. Ce type de serveur permet aussi de fournir l'adresse de la prochaine case à un serveur dont la case éclate.
Le serveur de noms est lui aussi basé sur le modèle *multi-thread* avec des threads d'écoute et des threads de travail. Il a une structure semblable à celle d'un serveur de données. Son architecture est décrite notamment dans [DL01]. Le serveur de noms est implémenté selon le modèle *client-serveur* où les clients et les serveurs de données jouent le rôle de clients soumettant leurs requêtes au serveur de noms.

3.2 Architecture d'une Case de Données dans SDDS-2005

Une case SDDS est stockée en mémoire centrale en tant que *fichier mappé* qu'offre Windows NT et Windows 2000. Les méthodes de communication inter processus fournies par l'API WIN32 reposent sur cette approche. Elles sont employées par SDDS-2005.
Cette organisation augmente l'efficacité des accès et génère d'importantes économies de temps CPU. La Figure 3- 4 montre la structure générale d'une case de données d'un serveur SDDS-2005.

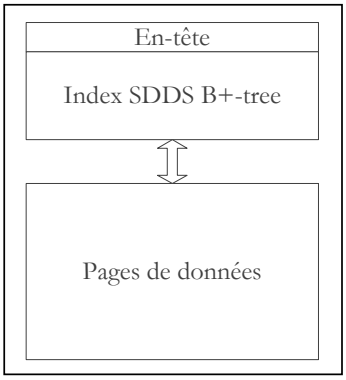


Figure 3- 4 *:* Organisation Interne d'une case SDDS

L'espace de stockage alloué à la case est répartie en trois zones (figure 3-4) [DNB00, DL01]:

− **Zone de l'en-tête :** L'en-tête contient, l'intervalle de la case (C_{min}, C_{max}], sa taille, le nombre courant d'enregistrements, l'adresse de la racine de l'index ainsi que la taille de ce dernier.

− **Zone de l'index :** C'est une variante originale d'un arbre B+ appelé SDDS *B+-Tree*. Les arbres B+, basés sur les arbres B, proposés en 1972 par Bayer et McCreight, sont aujourd'hui parmi les structures de données les plus utilisées. Voir notamment [K73, C79, KS86, VBW98]. La conception de SDDS *B+-Tree* est spécifique. D'une part, à son utilisation en RAM, pendant que la structure générique avait été conçue pour les disques. Mais surtout, sa conception est orientée vers l'efficacité de l'éclatement. Elle permet d'envoyer, plus facilement, la moitié droite de l'arbre, avec toutes ces feuilles, à la nouvelle case. Cette moitié est transformée en nouvel SDDS *B+-Tree* à part entière. Cette technique est bien plus rapide pour la réalisation d'un éclatement que l'extraction et l'envoi naïf des enregistrements individuels. Ces deux parties correspondent à la partie supérieure dans la figure 3-6.

− **Zone de page de données :** Cette zone contient les feuilles de SDDS *B+-Tree* avec les enregistrements (Figure 3- 6). Chaque feuille est une liste chaînée d'enregistrements, ordonnée par les clés, que l'on appelle aussi une *page* de données. Elle est composée d'un en-tête suivi d'une zone de stockage. Dans l'en-tête, on retrouve différents paramètres notamment la clé maximale de la page. Sa présence est nécessaire pour éviter un parcours inutile de la liste des enregistrements contenus dans cette page. En interne, chaque enregistrement est composé d'un en-tête suivi d'une zone de stockage appelée zone de données. Dans cet en-tête, on retrouve des paramètres désignant la longueur de l'enregistrement, le pointeur vers l'enregistrement suivant et la clé de l'enregistrement (Figure 3- 5). Les enregistrements ont une taille variable. Le champ de données peut être textuel ou binaire, des images par exemple.

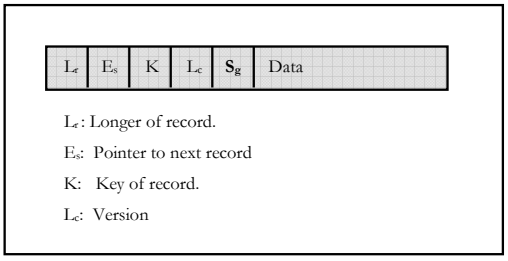


Figure 3- 5 : Structure d'un enregistrement dans SDDS-2005.

L'éclatement d'une page de données est similaire à celui d'un nœud interne de l'index. Il consiste à initialiser une nouvelle page à la fin de la zone de stockage des données ; puis à transférer la moitié des enregistrements de la page débordée vers la nouvelle page.

Nous utilisons deux fichiers *mappés* pour le stockage d'une case *SDDS*. Un fichier pour les zones d'en-tête et de l'index et un autre fichier pour la zone des données. L'utilisation de deux fichiers permet de gérer plus dynamiquement et donc efficacement, l'allocation de l'espace pour l'index et la zone des pages.

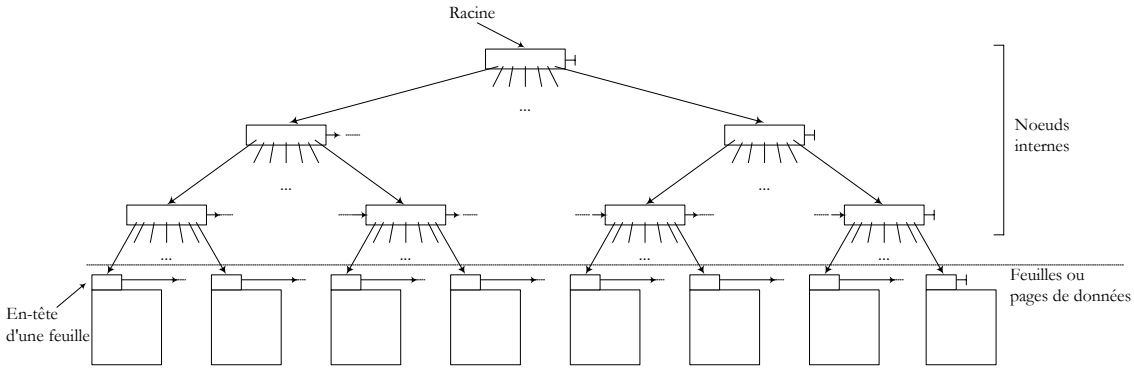


Figure 3- 6 : Architecture Générale de SDDS B+ Tree

3.3 Interaction Client / Serveurs de Données

Cette interaction concerne les différentes primitives du gestionnaire de SDDS que nous proposons. Nous décrivons brièvement les principales fonctions, leurs rôles dans la gestion et traitement de données ainsi que le principe de leurs fonctionnements.

3.3.1 Fonctions SDDS

Dans SDDS-2005, on retrouve deux niveaux de fonctions distinctes :

• Interface de commande de SDDS-2005 : Ces fonctions constituent les services que les clients SDDS offrent aux applications/usagers. Il s'agit de commandes de manipulation de fichiers et des enregistrements : créer un fichier, rechercher un ou plusieurs enregistrements, insérer un ou plusieurs enregistrements…Nous introduisons les commandes informellement ci-dessous. La description détaillée est dans [C05], reproduite dans l'Annexe A.

• Fonctions systèmes : Ces fonctions permettent d'exécuter les commandes. On les retrouve sur le client et sur le serveur.

3.3.2 Interface de commande

Nous décrivons brièvement les principales fonctions de SDDS-2005 La montre Figure 3- 7 les différentes opérations possibles que les clients SDDS ont à leurs disposition pour la manipulation des données d'une case SDDS.

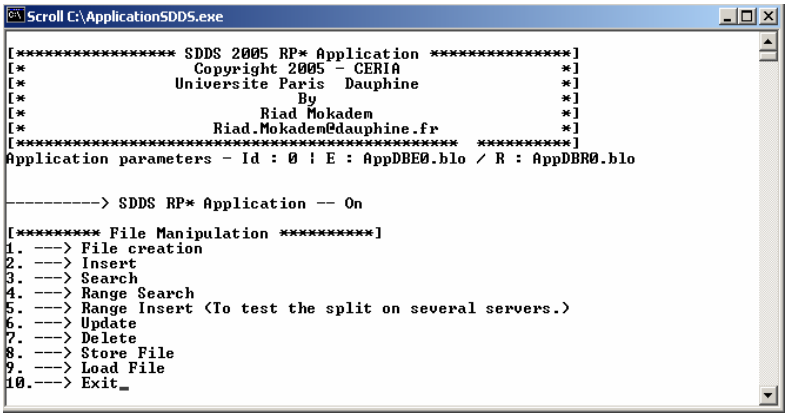


Figure 3- 7 : Interface de commande dans SDDS-2005.

3.3.2.1. Création de fichier

L'application fournit le nom de fichier et l'adresse IP du premier serveur. Le client SDDS vérifie, avec le serveur de noms, que le nom proposé n'est pas déjà utilisé. Un message est d'abord envoyé au serveur. Si le fichier n'existe pas encore, celui ci envoie un message au serveur de noms pour validation du nom de fichier, les paramètres de ce fichier sont alors enregistrés. La case est alors créé et un acquittement est envoyé au client. Si le nom existe déjà, le serveur de nom l'indique au serveur en question qui le signale, à son tour, au client concerné par la requête

3.3.2.2. Insertion d'enregistrements

L'insertion commence par l'encodage du contenu de l'enregistrement inséré. Nous abordons l'encodage de données dans la section 3-6 et plus en détail dans le chapitre 6. Deux types d'insertions sont possibles dans SDDS-2005. D'abord l'***insertion Simple.*** Si la taille de la requête est supérieure à 64Ko (en-tête et données), le protocole TCP est utilisé pour le transfert des données. Dans le cas contraire, un message UDP est suffisant pour contenir la requête et l'enregistrement inséré. De même

pour la taille de la réponse du serveur concerné. Dans le cas d'une image incorrecte au niveau du serveur, celui ci redirige la requête vers les autres serveurs grâce à un message multicast. Seul le serveur correct répond au client en lui envoyant un acquittement. A la réception de ce message, le client récupère l'*IAM* et corrige l'image du fichier.

On trouve également l'***insertion par intervalle***. Dans ce type de requête, le client envoie un message multicast à tous les serveurs. Chaque serveur ouvre d'abord un port TCP, puis répond à la requête en expédiant un acquittement au client. Le client collecte toutes les réponses, puis établit des connexions TCP concurrentes aux serveurs, pour la réception des données. Chaque connexion est maintenue par un thread de réception (d'écoute) au niveau du client. La requête se termine lorsque l'union des intervalles recouvre l'intervalle initial de celle ci. Un contrôle du flux est mis en place dans SDDS-2005. Il est optionnel. Il doit être demandé explicitement par le client. On discute le contrôle de flux dans les sections suivantes.

3.3.2.3. Recherche de données

On distingue dans SDDS-2005 la *recherche par clé* [DL01] et la *recherche non-clé* dite aussi *par le contenu* [LMS05a]. Ce dernier type ne figure pas dans le premier prototype SDDS-2000.

Recherche par clé : On recherche l'enregistrement ayant la clé recherchée par le client. Il ne peut y avoir qu'un seul par fichier. Les duplicata ne sont pas admis dans la version actuelle. Au niveau de fonctions système, la recherche se déroule de diverses manières, selon la variante RP* employée et selon la taille de l'enregistrement trouvé.

RP$*_N$*

Une requête *KeySearchRequest* est envoyée en multicast à tous les serveurs. Chaque serveur compare la clé à son intervalles Le serveur correct, dont l'intervalle contient la valeur de la clé, recherche dans sa case. Il envoie une réponse en unicast au client, positive ou négative.

RP$*_C$*

Le client envoie la requête de recherche par unicast ou multicast, selon l'image et la version de RP_c employée. Le multicast est alors traité par les serveurs comme pour $RP*_N$. Le serveur receveur d'un message unicast recherche sa case si la clé est dans son intervalle. Autrement il redirige la requête par multicast. Seul le serveur concerné répond au client. Celui ci reçoit l'enregistrement si il a été trouvé, sinon il reçoit un message informel indiquant que la recherche avait été sans succès.

Dans le cas positif de la recherche, si la taille de la réponse ne dépasse 64 KO, le serveur utilise UDP, pour la rapidité. Autrement, il emploie le TCP/IP.

Recherche par intervalle : Dans ce type de recherche, le client envoie plutôt un intervalle de clés à rechercher sut les différentes cases du fichier SDDS.

Recherche par le contenu : Cette fonction offre sous SDDS-2005 différents types de recherche de chaînes de caractères (ang. string or pattern matching). Elles se basent sur l'utilisation de signatures algébriques, celles cumulatives pour la plupart de ces opérations. La chaîne recherchée est encodée puis envoyée aux serveurs de données. L'encodage est alors transparent pour les serveurs de données. Nous traitons ce sujet dans les Chapitres 4 et 7. On retrouve cinq types de recherche par le contenu dans nos enregistrements :

- Recherche complète (utilisant les signatures algébriques). On recherche tout enregistrement dont la zone (champ) non-clé est égale à la chaîne donnée par l'application.
- Par préfixe : On recherche tout enregistrement dont le préfixe de la zone (champ) non-clé est égal à la chaîne donnée par l'application.
- Par chaîne partielle : On recherche les enregistrements dont la partie non clé contiens la chaîne envoyée.
- Par le plus grand préfixe commun : On recherche le plus grand préfixe dans la partie non clé des enregistrements.
- Par la plus grande chaîne commune : On recherche la plus grande chaîne commune entre la chaîne envoyée et la partie non clé de tous les enregistrements.

Par ailleurs, nous avons implémenté un nouvel algorithme de recherche de chaîne basé également sur les signatures cumulatives, la recherche par la méthode *n*-Gramme. Nous détaillons cet algorithme dans le chapitre 7.

3.3.2.4. Requête de Suppression

Le client peut demander la suppression d'un enregistrement en fournissant la clé de ce dernier. Sous SDDS-2005, au niveau système, on modifie le chaînage des enregistrements insérés, Figure 3- 8. La suppression est seulement logique, c'est à dire, que l'enregistrement supprimé est juste enlevé de la liste chaînée. Il reste cependant présent dans la case. Il est aussi inscrit sur la *liste des enregistrements supprimés*. Celle-ci est destinée à un module futur de réorganisation du fichier. On prévoit également de concevoir

une commande de récupération d'enregistrements (ang. Undelete). Cette fonction est surtout utile lors d'une suppression par erreur.

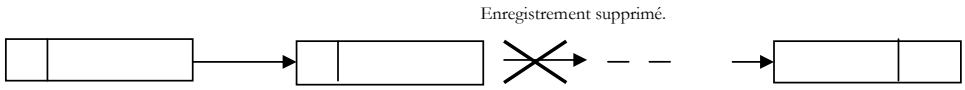


Figure 3- 8: Suppression d'un enregistrement dans SDDS-2005

3.3.2.5. Requête de Mise à jour

Cette fonction permet la modification de données non-clés d'un enregistrement indiqué dans la requête en question par sa clé. La mise à jour peut être *aveugle* ou *normale*. Pour ce dernier cas, au niveau de fonctions système, on manipule des images *avant* et *arrière* des enregistrements entre le client et le serveur. Différents clients peuvent mettre à jour le contenu d'un enregistrement en concurrence. On utilise des signatures algébriques pour le contrôle optimiste, sans verrouillage, donc plus performant d'habitude pour un fichier. Nous étudions les détails dans le Chapitre 5

3.3.2.6. Sauvegarde de fichier SDDS sur disque

Un fichier dont l'accès n'est pas très fréquent doit pouvoir libérer de la mémoire. Celle ci doit être réutilisable pour d'autres fichiers. En conséquence, il est possible que le client demande la sauvegarde d'un fichier SDDS sur le disque du serveur correspondant. Cette sauvegarde peut être faite avec ou sans libération de mémoire (seul ce dernier cas est activé au niveau SDDS-2005). L'utilisation de signatures algébriques permet de ne mettre sur disque que les données modifiées depuis la dernière sauvegarde. Cela est détaillé dans le chapitre 5.

3.3.2.7. Restauration d'un fichier SDDS à partir du disque

Si la présence d'un fichier sauvé sur disque redevient indisponible en mémoire, le client lance une requête de chargement de ce fichier à partir du disque des différents serveurs contenant ce fichier [M02]. Pour faire parvenir cette requête aux serveurs de données, le client procède de la même manière qu'une requête de sauvegarde. Il envoie un message multicast à l'ensemble des sites serveurs de données. Ces derniers lancent le chargement de ce fichier (s'il existe) en mémoire. Les serveurs ne possédant pas de fichier de même nom ignorent cette requête. Les autres la traitent en parallèle. A la fin, chaque serveur envoie l'intervalle de la case récupérée au client.

3.3.3 Fonctions systèmes

Elles interactent par les processus légers et les files d'attente. La commande de création d'un nouveau fichier déclenche l'initialisation d'une nouvelle case par de fonctions appropriées sur le serveur, l'appel au serveur de nom par ce dernier puis le retour avec l'ack positif ou négatif vers le client. La Figure 3- 9 montre le traitement des requêtes du client au niveau d'un serveur de données.

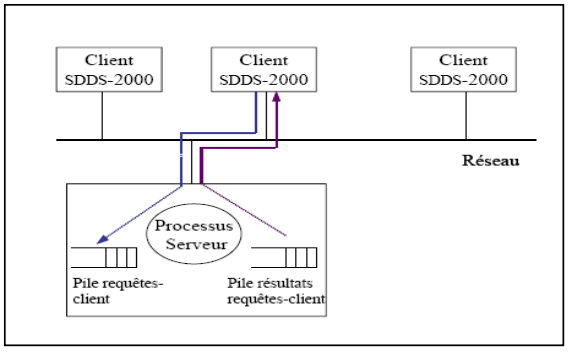


Figure 3- 9 : Traitement des requetes au serveur de données

De même, une commande d'insertion, peut déclencher la fonction système d'éclatement sur le serveur, appelant celle de la recherche d'un nouveau serveur, puis celle de la création de nouvelle case… etc. Ces interactions sont fort complexes. Voir notamment [D01] et [B00a] pour les détails sur lesquels notre implémentation en cours se base.

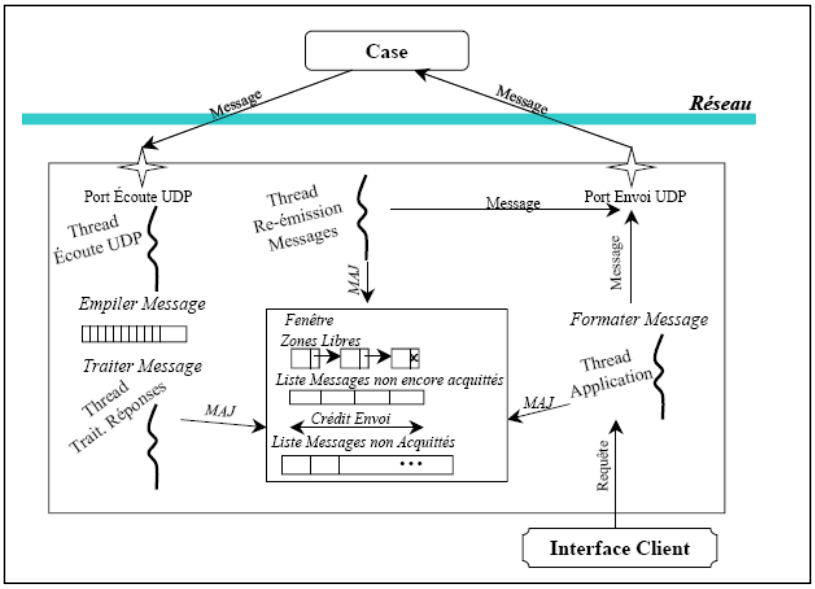


Figure 3- 10 : Fonctions systèmes d'envoi et de réception de meessages au client

La Figure 3- 10 représente le module d'envoi et de réception d'une requête au niveau du client. Celui ci rajoute cette requête dans une fenêtre des émission (cela est traité dans la section 3.6). Puis, il constitue le message de la requête et l'envoie au serveur de données à travers le port d'envoi UDP. Il se met en suite en attente d'acquittements par le serveur.
Dans le module de réception, un thread d'écoute UDP scrute le port correspondant attendant de recevoir la réponse à cette requête Un thread de traitement de réponse s'occupe ensuite (en cas de réception) de parvenir le résultat de la requête à l'application correspondante.

3.4 La Sécurité de données dans SDDS-2005

L'*encodage* est une nouvelle fonctionnalité dans SDDS-2005. Il permet une protection contre les vues accidentelles du contenu des enregistrements insérés sur les serveurs de données. Cet encodage se fait au client et est complètement transparent aux serveurs de données. L'opération inverse, le *décodage* se fait également au niveau du client lors d'une réponse à une requête de recherche par exemple. Les enregistrements contiennent les valeurs de signatures de leurs contenus initiaux. Néanmoins, un hacker expérimenté pourra décodifier les données. Tout un chapitre (chapitre 6) est consacré aux signatures algébriques et cumulatives et la protection contre les vues accidentelles des données.

3.5 Le Contrôle de flux et de perte de messages dans SDDS

Dans SDDS-2005, un contrôle du flux est mis en place pour les insertions massives. Il évite une surcharge du récepteur, surtout dans le cas d'un éclatement en cours d'insertions massives. En effet, la continuation des insertions dans le serveur qui devient incorrect pour celles-ci après l'éclatement, générerait un grand nombre de redirections inutiles.
Dans ce contexte, un contrôle du flux, optionnel, a été introduit. Il doit être demandé explicitement par le client. Notre mécanisme de contrôle de flux se base sur l'utilisation d'une fenêtre coulissante. L'approche s'inspire du mécanisme similaire proposé par *Van Jacobson* pour le contrôle de congestion dans le TCP/IP [J88]. Un système de *buffers* du client permettant d'enregistrer les requêtes à émettre est également utilisé. Ce système est composé d'un nombre fixé de zones pouvant stocker chaque requête non encore acquittée. Celles ci sont d'abord enregistrées avant d'être envoyées aux serveurs. A la réception des acquittements, elles sont supprimées du *buffer*.

Par ailleurs, Une requête émise par un client peut être perdue à cause du bruit sur le réseau de transmission. La réponse du serveur peut également être perdue. Aussi, la requête peut bien arriver à

destination, mais le *ListenThread* peut être indisponible au même moment, suite un engorgement du serveur. De ce fait, beaucoup de protocoles arment un temporisateur à l'envoi d'un message et lors de l'attente d'une réponse ou d'un acquittement de réception [T95]. Une méthode simple consiste à utiliser un *time out* pour borner le temps d'attente. A l'expiration de ce délai, le processus récepteur termine l'attente puis génère un code d'erreur. Pour s'assurer qu'un datagramme est bien arrivé à destination, l'émetteur doit demander un acquittement au récepteur. Ainsi, les serveurs envoient des acquittements explicites, individuels, aux requêtes des clients. Si un acquittement n'est pas reçu jusqu'à l'expiration du *time out*, une stratégie consiste à réémettre le datagramme original.

3.6 Protocoles de Communications dans SDDS-2005

Le prototype SDDS-2005 s'appuie sur le protocole TCP/IP. Celui ci assure l'interopérabilité entre une grande variété de plates formes et de technologies réseaux. Ce protocole fait référence à toute une famille de protocoles à l'intérieur de laquelle se retrouvent TCP et UDP. Ces derniers sont les principaux protocoles de la couche de transport. Ils se basent sur le protocole IP (couche réseau).
TCP assure la transmission par flot, fiable et en mode connecté. UDP assure la transmission par message, non fiable et en mode non connecté. Outre sa normalisation et sa standardisation universelle, le protocole TCP/IP est idéal pour l'adressage des éléments d'architectures distribuées.

3.6.1 Types de Messages

On distingue deux types de messages échangés entre les différents composants de SDDS-2005 : les messages de données et les messages de service. Ces types englobent plusieurs sous-types, comme le montre la Figure 3- 11
Les messages de service sont utilisés lors de l'initialisation d'un client, d'un serveur de données, du serveur de noms ou encore d'une case de parité. Ils sont aussi utilisés lors de l'allocation d'un site pour l'éclatement d'une case ou lors du contrôle de flux.

Les messages de données font référence aux messages utilisés pour l'accès aux données distribuées (requêtes des clients, réponses des serveurs) ou lors du transfert de données lors d'un éclatement d'une case de données.

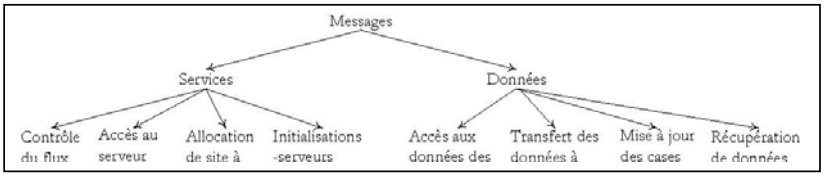


Figure 3- 11: Hiérarchie des messages dans SDDS-2005

Chaque type de message est identifié de manière unique. Il est composé d'un en-tête au début et d'une zone de données. Une procédure de décodage appropriée permet au récepteur d'interpréter l'en-tête pour identifier le type de message et éventuellement accéder aux données.

3.6.2 Protocole de Communication dans SDDS-CP

SDDS-CP est bâti au-dessus de la couche TCP/IP (Figure 3- 12). Il constitue le protocole de communication entre les différents composants SDDS-2005. Il est basé sur UDP pour les messages courts (transmissions non sécurisées) et TCP pour de longs messages dépassant 64 KO. UDP est un protocole plus simple et plus rapide que TCP. Il n'y a pas d'acquittement des paquets et donc plus de réémission ni de contrôle de congestion. Ce sera aux applications supérieures de gérer les pertes de paquets. Tout ceci permet d'accélérer le processus de communication. UDP convient bien aux applications de haute performance, vidéos, temps réel, etc. SDDS-CP fonctionne suivant le modèle client/serveur. Un ensemble de processus coopérants appelés *serveurs* offrent des services à des processus utilisateurs appelés *clients*. Chaque service est identifié par la combinaison de son adresse IP et de son numéro de port. Le client envoie une demande au serveur pour obtenir un service. Le serveur traite la demande puis envoie une réponse au client. Cette réponse contient les données demandées ou un code d'erreur indiquant pourquoi le service n'a pu être rendu.

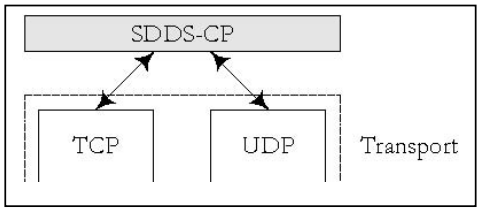


Figure 3- 12 : Positionnement de SDDS-CP.

3.6.3 Notion de Port

Un port fournit un emplacement pour envoyer des messages. La notion de port a été introduite pour assurer le multiplexage des communications entre applications. Un port fonctionne comme une file d'attente de message multiplexée, signifiant qu'il peut recevoir à la fois des messages provenant de sources multiples.

En général, un port est associé à une application et celle-ci ne peut être associée à un port déjà utilisé (sauf demande explicite de l'application qui le détient). La notion de port est utilisée distinctivement à la fois par TCP et UDP. Ainsi, nous parlons de *port UDP* et de *port TPC*.

3.7 La Communication entre Processus dans SDDS-2005

Les communications entre les composants SDDS-2005 sont basées sur le modèle des *Sockets*. Les données résident dans la mémoire vive de chaque serveur de données en tant que fichiers mappés dont a parlé plus haut. La communication locale et distante entre les processus légers de SDDS-2005 passe par ces fichiers et les sockets. Nous donnons ci-dessous un bref aperçu de l'utilisation de ces concepts dans notre implémentation.

3.7.1 Communication par les Sockets

Les sockets [S96] [S98] constituent un mécanisme réseau standard dont l'origine remonte à l'Unix de Berkley. L'évolution vers des mécanismes de communication inter processus dans l'environnement Windows a donné naissance aux *Windows Sockets*. Depuis, ils sont très utilisés dans la programmation réseau.

Un socket est caractérisé par une adresse Internet et un numéro port sur la machine locale. Une paire de sockets, une pour la machine de client, l'autre pour la machine serveur, définit la communication entre le serveur et le client identifiant un processus ou une application de manière unique sur le réseau.

Ils offrent un service de communication avec connexion (socket séquence de données) dans le cas d'une communication orienté flux de données ou en mode non connecté pour une connexion orientée messages (socket data gramme) en utilisant le protocole UDP.

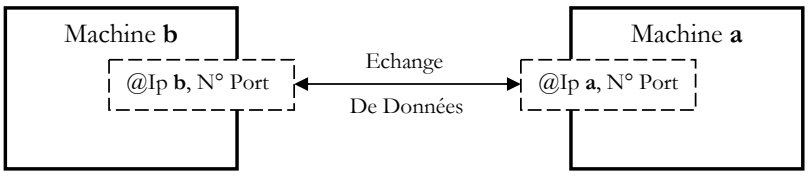


Figure 3- 13: Communication entre deux machines par sockets

3.7.2 Appel de procédure distante RPC (Remote Procedure Call)

Un processus de Windows NT possède son propre espace d'adressage. Il n'est pas alors possible d'appeler directement une procédure d'un autre processus. Les moyens de communication entre programmes de machines distantes sont fournis par les RPC [B92, S96]. Un programme qui appelle une fonction exécutée par un autre programme sur la même machine utilise un appel *LPC* (ang. Local Procedure Call). Dans SDDS-2005, les appels *LPC* sont utilisés lors de la communication entre le client et l'application.

3.7.3 Utilisation des Fichiers mappés

Windows NT se caractérise par une grande protection de la mémoire. Un processus n'a aucun moyen d'accéder aux ressources ou données d'un autre processus d'où la difficulté de partage de données. Cela se fait dans Win NT à l'aide de fichiers mappés [S96]. La technique de fichiers mappés se contente de correspondre au fichier une plage d'adresses mémoire dans l'espace d'adressage virtuel de l'application. Une vue du fichier est ensuite crée. L'accès à ce fichier se fera alors à travers un pointeur. Les autres threads accèdent à ce pointeur en appelant la fonction *OpenFileMapping ( )*.

La technique des fichiers mappés permet ainsi aux threads du processus d'accéder simultanément au contenu de cet espace d'adressage. Par ailleurs, Win 32 fournit une option permettant de forcer le stockage en mémoire centrale. La différence avec la mémoire virtuelle est que l'allocation de mémoire physique ne se fait qu'après l'appel à la fonction *ViewMapOfFile*.

3.7.3.1. Fichiers Mappés dans SDDS-2005

Les données de SDDS-2005 étant traitées en RAM, l'accès à un fichier mappé est beaucoup plus rapide que s'il était sur disque. Dans ce dernier cas, classique, pour lire le fichier, on est amené à allouer un tampon, ouvrir le fichier puis appeler une fonction de lecture pour copier le fichier totalement ou partiellement dans le tampon. Cela constitue une perte de temps très importante. Dans SDDS-2005, une case du serveur est associée, à l'initialisation, à un pointeur (handle). Celui ci désignera un fichier mappé créé par le biais de la fonction *CreateFileMapping*. Lorsqu'un serveur reçoit différentes requêtes provenant des clients, celles ci sont associées à différents processus qui s'exécutent d'une façon parallèle. L'accès au fichier est établit en utilisant la fonction *OpenFileMapping*. Chacun des processus peut avoir une vue en utilisant *MapViewOfFile*.

Dans le cas où plusieurs processus se partagent la même vue, ils se partagent forcément une partie de la mémoire d'où la nécessité de l'utilisation d'objets de synchronisations pour garantir l'intégrité de cette mémoire partagée.

3.7.4 Programmation Concurrente dans SDDS-2005

Dans toute application, le coût d'exécution d'une opération notamment, la réponse à une requête d'un client, doit être minimisé au maximum. Pour cela, la technique de *Multithreading* est utilisée (Figure 3-14). Elle permet de faire tourner plusieurs threads en même temps dans un même espace d'adressage. Une application serveur conçue pour servir un grand nombre de clients peut utiliser un *thread d'écoute* qui se charge de mettre les requêtes clients reçues dans une file d'attente et chaque requête est traitée par le prochain *thread de travail* disponible. En conséquence, des mécanismes de synchronisation sont utilisés notamment les évènements et les sections critiques. Un haut niveau de concurrence est également assuré grâce à des techniques d'ordonnancement tels que les files d'attentes.

Lorsqu'une application WIN32 est activée, un processus est créé. Mais, un processus n'existe jamais tout seul. En même temps, se crée un processus léger. [S96, R97]. Ce concept est un fait l'une de traductions possibles du concept de « thread ». On le définit aussi comme : « Unité d'exécution ordonnancé dans un processus » [S96] ou « Partie dynamique d'une application » [S96]. Dans WindowsNT, un processus peut contenir plusieurs processus légers qui s'exécutent de manière quasi parallèle comme si c'était des processus distincts, tout en partageant certaines ressources (variables globales du processus, espace d'adressage virtuel…). En revanche, chacun de ces processus légers dispose de sa propre pile d'appel, son état du CPU et son espace de stockage local.

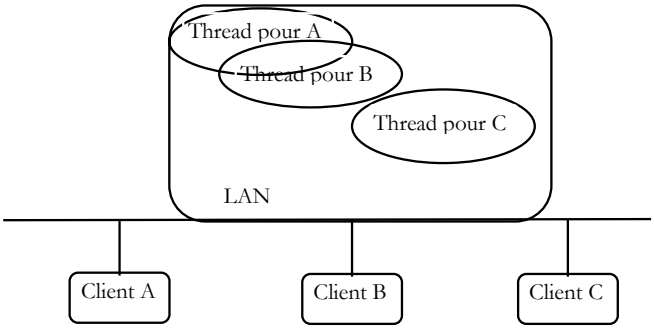


Figure 3- 14 : Le Multi-Threading

3.8 Synchronisation inter-processus et intra-processus

Dans un système multiprocesseur comme Windows NT, un processus peut avoir besoin de synchroniser ses activités avec d'autres processus (y compris les processus fils). De même, les *threads* d'un processus peuvent synchroniser leurs activités entre eux pour maintenir l'intégrité des ressources. Ces besoins de synchronisation sont importants spécialement pour les applications serveurs qui contiennent plusieurs *threads*. Les traitements sont aussi concurrents. Cela provoque des problèmes de *synchronisations* dues aux différents accès aux ressources (fichier, périphériques, zone de mémoire..) d'où la nécessité d'une bonne création et destruction de ces threads. Pour synchroniser différents accès à ces ressources et assurer l'exclusion mutuelle, Windows NT, comme Windows 2000, offrent plusieurs primitives (4 mécanismes) de synchronisation: *Evènements*, *sémaphores*, *mutexes* et *sections critiques*.

La section critique est la plus simple, elle est utilisée à l'intérieur d'un même processus et permet à une ressource globale d'être partagé par plusieurs threads. Les 3 autres permettent la synchronisation de *threads* situés dans des processus différents : le *Mutex* autorise un accès exclusif à une ressource par un *thread*; le *Sémaphore* contrôle le nombre de *threads* utilisant simultanément une même ressource; enfin, l'*Evénement* (*Event*) permet d'avertir un ou plusieurs *threads* de façon qu'ils effectuent une action donnée.

Par ailleurs, un objet SWMRG (Single Writer/Multiple Reader Guard) constitue une bonne solution pour les problèmes de synchronisation Il se compose d'objets de synchronisation comme les sémaphores, les mutexes et les évènements. Ainsi, dans notre application, un client ne pourra pas lancer une opération d'écriture et de stockage en même temps.

3.9 Synthèse

Dans ce chapitre, nous avons présenté notre prototype SDDS-2005. Nous avons décrit son architecture générale ainsi que celle de ces différents composants. Nous avons présenté les fonctions de l'interface de commandes et celles du système assurant l'exécution sous-jacente. Nous avons détaillé certains aspects du fonctionnement de notre prototype en ce qui concerne le contrôle de perte et de flux lors des communications. Puis, nous avons aussi décrit les techniques de programmation utilisées tels que l'utilisation des fichiers mappés pour nos structures de données ainsi que l'utilisation des sockets et de processus légers.

Chapitre

4 Les Signatures Algébriques

Dans ce chapitre, nous décrivons un nouveau type de signatures dites algébriques [LS04]. Celles ci présentent des propriétés semblables aux signatures bien connues, telles que le standard *SHA-1*[SH95] ou MD5. Ces signatures possèdent des propriétés qui nous servent pour décider de la similarité de deux pages, deux enregistrements ou encore de deux chaînes de caractères. La probabilité que deux pages etc. différentes aient une même signature est en effet presque nulle. Les signatures algébriques ont aussi des propriétés qu'on ne retrouve pas dans d'autres schémas de signatures. Celles-ci sont dues précisément à leur nature algébrique. Ce sont ces propriétés qui ont motivé le choix des signatures algébriques pour SDDS-2005. Dans notre prototype, cette technique est utilisée lors des opérations de sauvegarde de cases sur les disques, de mise à jour, de recherche de chaînes dans les enregistrements ou encore d'encodage des données afin qu'elles soient protégées contre une vue accidentelle sur les serveurs.

4.1 Signatures Algébriques

4.1.1 Concept de Signatures

Une signature numérique assure l'analogue d'une signature manuscrite d'un document sur papier. Il est difficile d'imiter une signature, si bien qu'un fichier accompagné d'une signature numérique est authentifié. Une signature numérique permet d'identifier un LDO (Large Data Object) pouvant être un enregistrement, une page de données ou un fichier de base de données. Cette identification est calculée à partir du contenu du LDO d'une manière telle que le résultat n'utilise que quelques Octets ou une vingtaine d'Octets, par exemple pour SHA-1 (SH95). Une des applications premières des signatures est la comparaison de fichiers afin de se prononcer sur leur similarité, de détecter une mise à jour ou enfin d'authentifier leur provenance [SNT04, TBB90]. Les signatures sont dès lors un outil idéal pour

comparer la concordance de réplication de fichiers par exemple. C'était d'ailleurs leur utilisation première.

Un « bon » algorithme de calcul de signature devait respecter plusieurs propriétés difficiles à atteindre. Il devait être d'abord rapide à calculer, avec la complexité de *O* (*N*) où *N* est la taille du LDO en symboles de calcul (les Octets notamment). Puis, il devait détecter les mises à jour courantes. Il s'agit tout particulièrement de permutations de symboles. Cette exigence élimine d'emblée les calculs triviaux tels qu'un simple XOR de tous les symboles du LDO.

Lors de la comparaison de fichiers, on peut notamment diviser ces fichiers en pages, puis calculer et comparer les signatures de ces pages, (Figure 4- 1). Le procédé permet de cerner les parties modifiées du fichier.

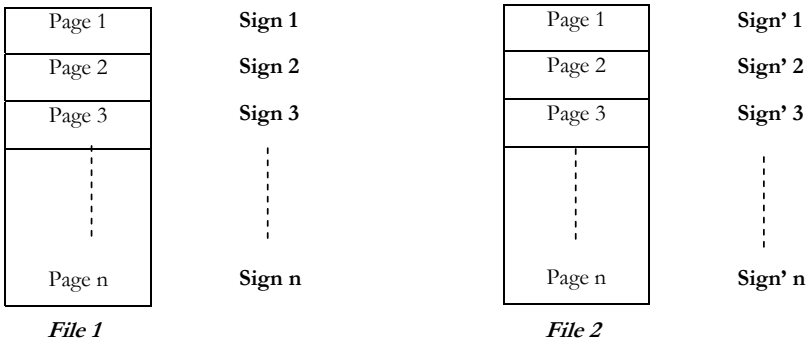


Figure 4- 1: Comparaison de fichiers à l'aide de signatures

Dans l'exemple ci dessus, on dira alors que deux fichiers *File1* et *File2* sont semblables s'ils possèdent les mêmes signatures.

$$\textbf{\textit{Sign}(File1)} = \textbf{\textit{Sign}(File2)} \quad \Rightarrow \quad \textbf{File1 = File2}$$

D'une manière plus générale, des formes spécifiques de signatures ont été proposées selon la fonction recherchée telle que la détection d'erreurs lors de transmission de documents ou l'altération malicieuse de données. Parmi les différents schémas, le SHA-1, déjà mentionné, est de loin le plus utilisé. Nous détaillons dès lors sa conception, à titre d'un exemple représentatif.

4.1.2 Les Signatures SHA-1

L'algorithme *Secure Hash Standard* (SHA) [SH95] constitue un outil de sécurité (cryptographie) permettant une authentification des documents à travers le Web. Le SHA-1 est le successeur du SHA-0 qui a été rapidement mis de côté par le NIST pour des raisons de sécurité insuffisante. Le SHA-0 s'est

vu modifié peu après sa sortie (1993) et complexifié pour obtenir le SHA-1 (1995). SHA-1 (Secure Hash Algorithm) est une fonction de hachage cryptographique conçue par la National Security Agency des États-Unis (NSA), et publiée par le gouvernement des États-Unis comme un standard fédéral de traitement de l'information. Les signatures cryptographiques sont basées sur des théories mathématiques. Une fonction de hachage[1] cryptographique est utilisée, entre autre, pour la génération de signatures électroniques. Elle rend également possible des mécanismes d'authentification par mot de passe sans stockage de ce dernier.

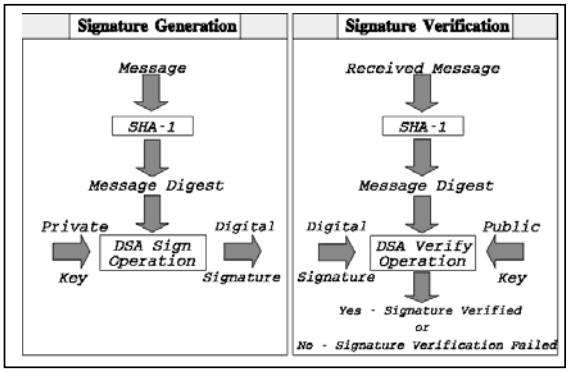


Figure 4- 2: Utilisation d'une signature SHA-1 pour produire la signature d'un message.

La fonction SHA-1, déjà implémenté dans Visual studio.net, produit un résultat (appelé « hash » ou condensé) de 160 bits (40 chiffres hexadécimaux). Ce message, appelé *Message Digest*, constitue par la suite une entrée pour une fonction appelée *Digital Signature* (). Celle ci permet alors de vérifier ou de générer la signature finale de la chaîne de caractères.

Le SHA-1 prend un message d'un maximum de 264 bits en entrée. Son fonctionnement est similaire à celui du MD4 ou MD5 de Ronald Rivest. Si le message n'a pas une longueur qui est un multiple de 512 alors l'algorithme rajoute un bit à 1 suivi d'une série de bits à 0. Finalement, la longueur du message (en bits) codée sur 64 bits est ajoutée à la fin de cette séquence. Quatre fonctions booléennes sont définies, elles prennent 3 mots de 32 bits en entrée et calculent un mot de 32 bits. Une fonction spécifique de rotation est également disponible, elle permet de déplacer les bits vers la gauche (le mouvement est circulaire et les bits reviennent à droite). Une de ces rotations n'était pas présente dans le SHA-0, elle permet de casser certaines caractéristiques linéaires dans la structure. Cela permet d'éviter une attaque sur les bits neutres par Eli Biham, technique reprise pour calculer la collision complète sur SHA-0 (Antoine Joux et al.).

[1] : Fonction de hachage: Fonction réduisant une liste de données de taille arbitraire en taille fixe et lui associe une valeur numérique.

Le SHA-1 travaille ensuite individuellement sur des blocs de 512 bits. L'algorithme calcule 80 rondes ("rounds") successives et applique une série de transformations sur l'entrée. La première étape consiste à calculer 80 valeurs sur 32 bits. Les 16 premières valeurs sont obtenues directement à partir du bloc « message » en entrée. Les 64 autres sont calculées successivement. Le SHA-1 les obtient grâce à une rotation (absente dans SHA-0) qui est appliquée sur le résultat d'un XOR. Il utilise pour cela 4 mots obtenus dans les itérations précédentes. On définit ensuite 5 variables qui sont initialisées avec des constantes (spécifiées par le standard), le SHA-1 utilise encore 4 autres constantes dans ses calculs. Si un bloc de 512 bits a déjà été calculé auparavant, les variables sont initialisées avec les valeurs obtenues à la fin du calcul sur le bloc précédent.

Il s'ensuit 80 tours qui alternent des rotations, des additions entre les variables et les constantes. Selon le numéro du tour, le SHA-1 utilise une des quatre fonctions booléennes. L'une de ces fonctions est appliquée sur 3 des 5 variables disponibles. Les variables sont mises à jour pour le tour suivant grâce à des permutations et une rotation. En résumé, le SHA-1 change sa méthode de calcul tous les 20 tours et utilise les sorties des tours précédents.

A la fin des 80 rondes, on additionne le résultat avec le vecteur initial et les cinq variables concaténées ($5 \cdot 32 = 160$ bits) représentent la signature. La figure 4-3 montre le processus de génération puis de vérification des signatures cryptographiques SHA-1.

Dans des applications relatives à la cryptographie et sécurité des données, la fonction SHA-1 permet le calcul du message digest. Celui ci, combiné avec une clé, permet de produire la signature finale du message.

Au décryptage de l'information, l'opération inverse est réalisée. La signature obtenue est comparée avec la signature sauvegardée. Cela permet la détection d'un changement au niveau du message initial. Dans notre application, SHA-1 produit une signature dite 'cryptographique' pour chaque page de données.

4.1.3 Introduction aux Signatures Algébriques

Différents travaux existent dans ce contexte. Plusieurs schémas pour le calcul de signatures algébriques ont été étudiés. On cite les travaux de Karp Rabin [KR87], Broder [B93], Schwarz & al [SBB90], Suel & al [SNT04]. Le calcul de signatures algébriques par notre approche s'appuie spécifiquement sur l'algèbre de Corps de Galois. Nous commençons par le rappel de notions de base correspondantes [LS04].

4.1.3.1. Calcul dans un Corps de Galois

Notre technique de signatures algébriques s'appuie sur le calcul dans un corps fini [SC83, E96]. Les corps finis sont aussi appelés *corps de Galois* en hommage au mathématicien Français *Evariste Galois* [1811-1832] qui les a définis. On note un corps de Galois par l'abréviation GF (ang. Galois Field). Un GF est muni de deux opérations internes, l'*addition* ayant la propriété d'être commutative et la *multiplication* qui est associative et distributive par rapport à la loi additive. Un GF possède aussi les éléments 0 et 1 avec les propriétés habituelles. On peut ainsi effectuer dans un GF les opérations de soustraction et de division.

Un corps de Galois est souvent noté GF (q^f), ayant les éléments 0, 1, 2,…, q^f -1. On appelle q la *caractéristique* du GF. Dans notre application, on utilise $q = 2$ comme nous détaillons plus loin. Tout GF est fermé pour l'addition et la multiplication. Le résultat de l'addition, soustraction, multiplication ou division de deux éléments du corps, est un élément du corps. Ainsi, si e_1 et e_2 appartiennent à *GF (q^f)*, alors la somme de $e1$ et $e2$ dans GF (q^f) est égale à la somme des deux entiers. Aussi, le produit de e_1 et e_2 dans GF (q^f) est égale au produit des deux entiers réduit modulo q^f. On dit alors que GF (q^f) est fermé pour l'addition et la multiplication.

Le plus petit corps de Galois est le corps binaire GF (2) = {0, 1}. Les éléments d'un corps de Galois sont souvent présentés en polynôme de la forme $c_0+c_1x+c_2x^2+\ldots+c_{n-1}x^{n-1}$ où n est égal à q^f et tous les c_i appartiennent à GF (2). La représentation polynomiale du $i^{\text{éme}}$ éléments de GF (q^f) est le reste de la division euclidienne de x^i par p(x), où p(x) est un polynôme irréductible (pas divisible par un autre polynôme de degré inférieur) de degré f.

A titre d'exemple, sachant que le polynôme $p(x) = x^3+x+1$ est irréductible dans GF (2^3), le énumère l'ensemble des éléments de GF (2^3).

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

Tableau 4- 1: Représentation polynomiale des éléments de GF (2^3)

L'addition de deux éléments du GF (q^f) est l'addition terme à terme. Par exemple, $x^2 + (x^2+1)$ est égal à 1. Il en est de même de la soustraction. Pour le produit de deux polynômes nous utilisons la propriété de fermeture du corps. Par exemple, pour calculer $x * x^2$, il faut trouver à quelle classe appartient x^3. Pour cela, on effectue la division euclidienne de x^3 par $p(x)$ qui donne $x+1$.

Une propriété importante d'un GF est l'existence d'un élément (polynôme) dit *primitif*, souvent noté α tels que tous les autres éléments non nuls sont des puissances de α. Plus précisément, on peu représenter tous les éléments du GF non nuls comme : $(\alpha, \alpha^1, \alpha^2, \ldots, \alpha^w)$ avec $w = q^f-2$. A titre d'exemple, nous montrons ainsi la représentation des éléments de GF (2^3), avec le calcul polynomial correspondant. Dans GF (2^3), seuls les éléments dont les puissances sont de degré inférieur à 3 sont retenus, ils correspondent à $\{1, \alpha, \alpha^2\}$

$\alpha^3 = \alpha+1$ (car α est une racine de $p(x)$: $\alpha^3 + \alpha+1 = 0$)

$\alpha^4 = \alpha * \alpha^3 = \alpha * (\alpha+1) = \alpha^2+\alpha$

$\alpha^5 = \alpha * \alpha^4 = \alpha * (\alpha^2+\alpha) = \alpha^3 + \alpha^2 = \alpha+1+\alpha^2$

$\alpha^6 = \alpha * \alpha^5 = \alpha * (\alpha^2+\alpha+1) = \alpha^3 + \alpha^2+\alpha = \alpha+1+\alpha^2+\alpha = 1+\alpha^2$

$\alpha^7 = \alpha * \alpha^6 = \alpha * (\alpha^2+1) = \alpha^3+\alpha = \alpha+1+\alpha = 1$

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

Cette propriété permet de choisir α (élément primitif) puis de déterminer, pour chaque élément g du champ, la puissance i telle que $\alpha^i = g$. On écrit alors :

$$\mathbf{i = log_{\alpha}(g)}$$

Inversement $$\mathbf{g = Antilog_{\alpha}(i)}$$

Les opérations sur les éléments d'un GF basés sur la représentation des éléments de GF en polynômes sont en général coûteuses en terme de mémoire et de CPU. L'addition dans un GF (2^f), que nous utilisons d'habitude, se calcule en pratique par l'opération '*XOR*' bien connue. Il en est de même de la soustraction. Ainsi on a :

g+b= g *XOR* b
g - b= g *XOR* b

Ensuite, une méthode plus commode pour les GF (2^f) avec $f=1,2$ est d'utiliser des tables dites *Log/Antilog*. Ces corps sont ceux que nous utilisons ci-dessous. La méthode emploi les propriétés discutées des éléments primitifs, en représentant chaque élément par la puissance adéquate de α

choisie. Les logarithmes et les antilogaritmes ci-dessus sont alors tabulés. La taille des tables est déterminée par celle du corps de Galois. En conséquence, la table des *Antilog* et *Log* auront 2^f entrées. Le Tableau 4- 2 montre un exemple de ces deux tables pour f=3.

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

Tableau 4- 2: Table *AntiLog* implémenté en mémoire GF (2^3)

Les opération de multiplication et de division dans GF (2^f) sont traitées comme suit :

g*b=*Antilog*$_p$ (*log*$_p$ (g) + *log*$_p$ (b) Mod 2^f -1)
g/b=*Antilog*$_p$ (*log*$_p$ (g)- *log*$_p$ (b) Mod 2^f-1)

L'utilisation du *Modulo* (2^f *–1)* est introduite à cause de l'addition de deux valeurs logarithmiques. On l'utilise pour se positionner à l'intérieur de la table *Antilog*. Cela ralentit le calcul des signatures algébriques. Pour éviter l'utilisation du '*Modulo*', une solution consiste à doubler la taille de la table Antilog. Celle ci aura alors 2^{2f-2} positions ce qui est suffisant pour ne pas avoir de débordement au delà de cette table. Cette approche a été adoptée pour le calcul de signatures dans SDDS-2005. L'utilisation du *Modulo* reste néanmoins nécessaire lors de la recherche de chaînes dépassant 2^f caractères. On aborde cela en détail dans la section réservée à la recherche par les signatures cumulatives dans le chapitre 7.

Suivant ce qui est décrit ci dessus, l'opération de multiplication d'un élément *left* par un élément *right* s'effectue de la manière suivante :

```
GFElement mult (GFElement left, GFElement right)
{
  if (left==0||right == 0) then return 0;
  Return Antilog [log[left]+log[right]];
}
```

En termes d'instructions machine, l'opération nécessite deux comparaisons, quatre additions (trois à partir de la table look-up), trois 'memory fetches' et un retour d'état [LS04].
Dans notre application, si on se base sur GF (2^{16}), chaque élément du corps, appelé symbole, est représenté par un mot de 16 bits (Tableau 4- 3). Un symbole est représenté alors sous forme polynomiale dont les coefficients représentent les bits de ce polynôme (par exemple 10010011

correspond à x^7+x^4+x+1). Remarquons les valeurs Log sont des entiers tandis que les valeurs Antilog sont des GF.

Nombre Binaire	Hexadécimal	Log_α	$Antilog_\alpha$
0000 0000 0000 0000	0	-	1
0000 0000 0000 0001	1	0	2
0000 0000 0000 0010	2	1	4
0000 0000 0000 0011	3	61481	8
0000 0000 0000 0100	4	2	16
0000 0000 0000 0101	5	57427	32
⋮	⋮	⋮	⋮
1111 1111 1111 1111	FFFF	4725	1

Tableau 4- 3:Table *Antilog* dans implémenté en mémoire GF (2^{16})

4.2 Définition Générale de Signatures Algébriques

On appelle ici *page P* de taille *l* une suite de symboles $p_1, p_2, \ldots p_i$; i=0, 1,.., l-1. Dans notre cas, chaque symbole p_i représente 1 ou 2 Octets suivant le type de caractères employés, ASCII ou Unicode par exemple. On assimile les symboles aux éléments de GF (2^f) avec f=8 ou f=16 respectivement, avec $l<2^f-1$. Soit α un élément primitif de GF (2^f). Soit $\boldsymbol{\alpha}= (\alpha^1, \alpha^2, \alpha^3,\ldots,\alpha^n)$, un vecteur non nul, avec n<<ord (a)=2^f-1. Alors, la signature $Sign_{\boldsymbol{\alpha},n}(P)$ de P sur n caractères, basée sur $\boldsymbol{\alpha}$, est un vecteur sur n éléments représenté [LMS03] comme suit :

La signature $Sign_{\boldsymbol{\alpha},n}(P)$ de P sur n caractères, basé sur $\boldsymbol{\alpha}$, est un vecteur sur n éléments représenté [LMS03] comme suit :

$$Sign_{\boldsymbol{\alpha},n}(P) = (sign_{\alpha}^{1}(P), sign_{\alpha}^{2}(P),\ldots, sign_{\alpha}^{n}(P)).$$

Pour chaque $\boldsymbol{\alpha}$, la signature $Sign_{\boldsymbol{\alpha}}(P)$ s'écrit :

$$Sign_{\alpha}(P)= \sum p_i \alpha^i \quad i=0,.., l$$

Comme nous montrons plus loin, plus n est grand dans $Sign_{\alpha,n}$ (P), moindre est la probabilité d'une *collision*. Une collision se produit si deux pages différentes possèdent une même signature. Les signatures algébriques possèdent par rapport à ce problème, deux propriétés cruciales, démontrées dans [LS04].

La première, introduite par ces signatures, permet d'atteindre la probabilité de collision nulle, dans les conditions spécifiques, paramétrables par la valeur de *n*. Plus précisément, pour toute page *P* constituée de *l* caractères avec $l < 2^f-1$, la signature $Sign_{\alpha,n}$ (P) détecte tout changement de *n* caractères au plus. En conséquence, on détecte sûrement la différence correspondante entre deux pages. On détecte aussi sûrement toute mise à jour d'au plus *n* symboles dans P.

La 2ème propriété concerne la probabilité de collision quand la différence peut dépasser *n*. Dans ce cas, cette probabilité est d'habitude de 2^{-fn}. Ce qui est la même valeur que pour les schémas de signature habituels, comme ceux déjà cités.

Dans notre cas, l'utilisation de f =8 puis 16 s'est avérée optimale, pour des raisons qui apparaîtront progressivement. D'une manière générale, pour f = 8, la taille limite d'une page pour profiter de la détection sûre de collisions est alors de 128 Octets (2^f). Pour f=16, cette limite devient presque de 255 Ko, donc bien plus pratique.

Il est aussi visible et montré dans [LS04] que la signature algébrique détecte en général une permutation de symboles. Intuitivement, la raison en est la multiplication par la puissance de α différente après le changement de place du symbole. Les propriétés discutées ainsi que la rapidité potentielle de calcul de ces signatures que nous discutons plus loin, montrent l'intérêt prospectif du concept pour les applications. Notamment, par le fait que l'on peut utiliser les signatures bien plus compactes que celles générées par SHA-1. L'examen des applications prospectives dans le cadre des SDDS a montré que les tailles de 1 à 4 Octets peuvent déjà suffire. Le calcul de signatures algébrique apparaît aussi visiblement plus rapide que SHA-1, pour les valeurs de *n* considérées, comme notre étude expérimentale confirme d'ailleurs. C'est l'ensemble de ces raisons qui nous a amène à intégrer cette technique dans SDDS 2005 pour des usages diverses. Il s'agit de fonctions de sauvegarde, de mise à jour et de recherches non-clé, détaillées dans ce qui suit.

Les signatures algébriques on par ailleurs bien d'autres propriétés algébriques utiles. Nous discutions ces propriétés dans les chapitres consacrés aux fonctions spécifiques mentionnées.

4.2.1 Calcul de signature dans SDDS-2005

Dans SDDS-2005, on calcule une signature algébrique pour une page (16Ko) de données lors de la sauvegarde d'un fichier. Cette signature est calculée pour chaque enregistrement (ou une partie d'enregistrement) dans le cas de mises à jour et de recherche. Dans le premier cas, on utilisera une signature algébrique à 2 symboles (4 Octets). Pour les mises à jour, une signature algébrique sur 2 Octets est utilisée. Enfin, on calcule de signatures sur 1 octet lors de la recherche en utilisant des

signatures algébriques spécifiques, dites cumulatives. On aborde cela en détail dans les chapitres respectifs.

Le calcul actuellement implémenté utilise les tables *Log/Antilog*. Plus précisément, soit P une page de données constituée de l symboles successifs $p_0 p_1 \ldots p_l$. Les symboles sont interprétés comme des valeurs logarithmiques comprises entre 0 et 2^f-1 (inclusivement). Pour $n = 1$, le calcul d'une signature algébrique se présente comme suit, en utilisant la notation habituelle de langage C :

```
For (i=0 ; i<=Page_size ;i++)
        Signature ^ =Antilog [i+ p i ];
```

Ici, le calcul concerne la signature $Sign_\alpha(P)$. Le signe ^ signifie l'opération '*XOR*'. La signature sur n symboles est représentée alors par la concaténation des n symboles calculés. La signature finale s'écrit : $Sign_{\alpha, n}(P) = [\ Sign_\alpha^1(P), Sign_\alpha^2(P), \ldots., Sign_\alpha^n(P)\ ]$.

Pour le calcul de $Sign_{\alpha, n}(P)$ pour n>1, la multiplication par α^k (k=1..n) se calcule de la manière suivante : *Signature* ^ =*Antilog* [(k*i) + p_i];

Dans SDDS-2005, les signatures algébriques sont calculées soit au niveau des serveurs de données (lors de la sauvegarde d'un fichier) ou au niveau des clients (lors d'une opération de mise à jour d'enregistrements). Différentes comparaisons interviennent à ce moment permettant la détection du moindre changement. Cela a pour conséquence la sauvegarde ou la mise à jour des enregistrements réellement modifiés. Ces comparaisons peuvent aussi renseigner sur la similarité entre deux chaînes comparées.

4.3 Optimisation de Calcul de Signatures Algébriques

Notre calcul de signatures algébriques est un choix parmi plusieurs variantes possibles. Nous présentons maintenant quelques variantes que nous avons également étudié. Ils n'ont pas été choisis pour diverses raisons. Ils restent néanmoins des sujets d'étude intéressants dans le cadre de travaux prospectifs.

4.3.1 Utilisation de Cache L1 et L2

Dans un serveur SDDS-2005, une case de données contient en général plusieurs pages de données. Des milliers de signatures algébriques sont alors calculées. Elles utilisent les mêmes tables *Log/Antilog*.

Pour accélérer le calcul, il semble utile de mettre l'une ou les deux tables dans les caches L1 et L2 du Pentium. Alternativement, si possible aussi y cacher la page courante. Cette approche est théoriquement possible par l'utilisation de l'instruction assembleur '*prefetch*'. Toutefois, nous n'avons pas encore mis en pratique l'appel au Prefetch sous le compilateur C de Microsoft que nous devions utiliser pour l'implémentation SDDS-2005. C'est la raison pour laquelle cette variante est restée prospective.

4.3.2 Schéma de Horner et Tables de multiplication par α

Le calcul d'une signature peut être modifié pour être aisément factorisé en schéma populaire de Horner, utilisé par exemple par Karp-Rabin [KR77]. Dans notre cas spécifique, ceci peut donner lieu au concept dit de *signature inversée*, notée S^I ci-dessous et défini par le schéma suivant, pour $i = 1..n$ ($i^{ème}$ symbole de la signature à n symboles) :

$S^I(P) = (p_1\alpha^i + p_2)\ \alpha^i + p_3)\ \alpha^i ... + p_{m-1})\ \alpha^i$

Le calcul de la signature inversée de cette manière, à la place de celle discutée jusque-là, était d'avantage compatible avec celui de signature cumulative que nous détaillons dans le Chapitre 7. Une propriété utile dans ce cadre était la possibilité de remplacement de la multiplication usitée par celle par celle utilisant les *tables de multiplication par* α^i. Plus précisément la table T tabule pour chaque symbole p du GF utilisé la valeur : $T(p) = \alpha^i p$. La multiplication est alors potentiellement plus rapide que par notre méthode de base.

On discute dans le Chapitre 8 quelques mesures de performances relatives à cette méthode. Celles ci montrent que cette méthode peut effectivement accélérer le calcul d'une manière importante (surtout les signatures cumulatives). Par contre, l'emploi de signatures inversées complique et semble ralentir les principaux algorithmes de recherches de chaînes encodées que nous avons implémenté. L'étude détaillée de cette approche est ainsi restée également parmi les objectifs futurs.

4.3.3 Tables de Broder

Il s'agit d'une méthode alternative pour la multiplication $\alpha^i p$, en conjonction avec le schéma de Horner. On utilise une boucles sur i décalages et 'XORs' de la valeur décalée avec p, avec une correction après certains décalages par la *table de Broder*, [B93]. Les signatures algébriques de base apparaissent dans l'ensemble plus avantageuses à l'heure actuelle pour nos besoins. L'étude approfondie de cette méthode peut être utile pour les travaux futurs. Elle pourrait être d'intérêt majeur

pour un GF (2^f), avec f = 32, 64. Les tables *Log/Antilog* ou T ci-dessus seraient alors d'une taille en général prohibitive.

4.4 Comparaisons et Synthèse

Nous avons présenté le concept de signature algébrique. Nous calculons une telle signature suivant un élément primitif du corps de Galois constitué à l'heure actuelle de 2^8 ou 2^{16} éléments. Les signatures algébriques ont des propriétés nouvelles par rapport aux schémas populaires, le standard *SHA-1*, notamment. Elles peuvent être plus discriminatives avec la probabilité de collision égale zéro. Elles peuvent aussi être bien plus compactes : 1 à 4 Octets au lieu de 20 pour SHA-1 ou 16 pour MD5. Enfin, elles présentent d'autres propriétés d'intérêt pour nous que nous discutons plus tard. C'est cet ensemble qui a motivé leur choix pour SDDS-2005.

Chapitre

5 *Signatures Algébriques pour la Sauvegarde de Données dans SDDS-2005*

La sauvegarde répartie permet une gestion dynamique de grandes masses de données. Elle étend aux structures scalables et distribuées, les techniques de sauvegarde classiques plus centralisées et limités par leur manque de disponibilité et de flexibilité. Dans ce chapitre, on décrit l'utilisation de signatures algébriques pour la sauvegarde d'un fichier SDDS. Grâce à ces signatures, on ne sauvegarde sur les disques que les données ayant été mises à jours depuis la dernière sauvegarde. Cette technique induit des gains de temps considérables de transfert de données vers le disque [MLS03].

5.1 Principes Généraux

Un fichier SDDS-2005 réside en RAM distribuée. Pour des raisons de vulnérabilité d'une RAM ou de partage de celles-ci entre plus de fichiers que la capacité de la RAM peut supporter pour les cases correspondantes, le client peut souhaiter sauver un fichier SDDS sur les disques locaux des serveurs. SDDS-2005 offre à cette fin la requête de sauvegarde. Celle-ci précise si la case doit rester en RAM après la sauvegarde, ou l'espace de ses fichiers mappés est rendu libre (sauvegarde avec évacuation de la case). Il peut être alors alloué à une autre case. La requête de sauvegarde est complémentée naturellement par la requête de restauration à travers le chargement de ce fichier suite à une requête dite de chargement [M02]. L'organigramme de la Figure 5- 1 montre le traitement global d'une requête de

sauvegarde. Nous commençons par celui sur le client, puis nous discutons le traitement sur les serveurs.

5.1.1 Types de Sauvegarde d'un Fichier SDDS

IL existe deux types de sauvegarde dans SDDS-2005. Le client précise ce type lors de la constitution de la requête afin de l'acheminer aux serveurs Ces deux possibilités sont: la sauvegarde sans libération de la mémoire et celle avec libération de mémoire.

Sauvegarde sans libération de la mémoire : Tout en maintenant le fichier en mémoire, une copie de celui ci est sauvegardée sur le disque. Ce type de sauvegarde permet surtout la récupération des données dans le cas d'une panne au niveau de serveurs de données.

Sauvegarde avec libération de la mémoire : Ce type de sauvegarde appelé aussi 'évacuation' permet la libération de la mémoire occupé par le fichier à sauvegarder et l'utilisation de son espace mémoire par d'autres fichiers. La sauvegarde d'un fichier peut être envisageable si ce dernier devient peu utilisable en mémoire et donc occupant inutilement de la place mémoire [M02, LSS02]

5.1.2 Traitement par le client

Le traitement de la requête de sauvegarde commence par sa diffusion par le client ayant reçu la commande, à tous les serveurs. Le client utilise un message *multicast* de type UDP, car le message est court. Il assemble néanmoins plusieurs paramètres. Il y a l'identifiant de l'application émettrice, le code de la requête (Code sauvegarde), l'algorithme utilisé ($RP*_c$ ou $RP*_n$), le fichier concerné et le bit d'option pour le contrôle de flux (si il est utilisé).

Le client se met ensuite à l'écoute des acquittements individuels des serveurs concernés. Il s'agit normalement des acquittements positifs, accompagnés par les paramètres reçus et l'intervalle de la case localement sauvegardée. Le client vérifie si tous serveurs concernés ont acquitté l'opération en procédant à l'union des intervalles. En d'autres termes, il suit le protocole de terminaison *déterministe*. Dans l'absence constatée de réponse d'un ou plusieurs serveurs, après un time-out, le client relance la requête.

5.2 Traitement par les Serveurs

A la réception d'une requête de sauvegarde, un serveur n'ayant pas de case appartenant au fichier, ignore la requête. Les autres appellent la fonction de sauvegarde. C'est cette fonction qui fait l'usage des signatures algébriques que nous détaillons plus loin.

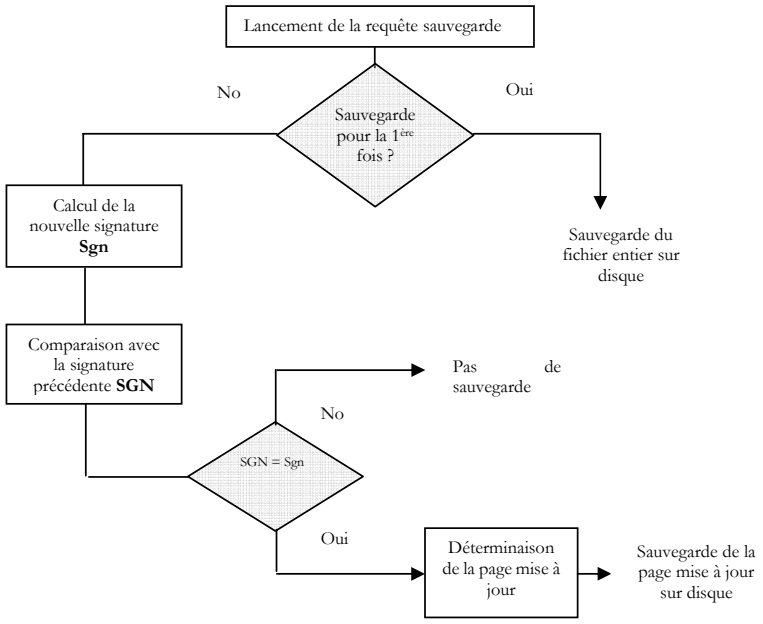


Figure 5- 1: Organigramme de déroulement d'une Requête de Sauvegarde dans SDDS-2005

La Figure 5-1 résume le déroulement de la sauvegarde. A la fin du traitement local, chaque serveur ayant sauvé la case émet vers le client un acquittement individuel positif décrit plus haut.

5.2.1 Partition en pages signées

La fonction de sauvegarde partitionne logiquement la case en pages de longueur fixe. La partition réelle concerne le fichier mappé des données d'une part et celui de l'index d'autre part. La longueur d'une page de données est actuellement de de 16 KOctets sauf pour la dernière page. Chaque page est logiquement munie d'une signature algébrique calculée par la fonction de sauvegarde.

Nous avons choisi le calcul dans GF (2^{16}) produisant la signature à 2 symboles. Ainsi, la probabilité de collision est d'habitude de 2^{-32}, donc négligeable en pratique. Le calcul concerne seulement les données non clés de chaque enregistrement. Les méta données (en têtes des feuilles de données) ne sont pas concernées par ce calcul puisque non déterminantes pour la détection d'une mise à jour. En effet, un enregistrement peut subir des modifications sur sa valeur sans mettre à jour sa taille…etc.

Les signatures sont sauvées dans une table dite *carte de la case* (ang. bucket map). La 1ère sauvegarde sauves toutes les pages. Ensuite, la page n'est écrite sur le disque que si sa nouvelle signature, dite *signatures_après*, diffère de celle précédente, dite *signatures_avant*, Figure 5- 1.

Il est utile de comparer ici notre approche avec celle usuelle. Cette dernière consisterait aussi à diviser chaque case en pages munis chacune d'un bit, dit souvent en anglais le '*dirty bit*'. On met à 0 ce bit lors de la sauvegarde de cette page sur disque et on le remet à '1' à la mise à jour constatée d'une page. Ce traitement alourdit à l'évidence le traitement en temps réel de mises à jour de la case qui peuvent être fréquentes dans une structure scalable. D'autant plus qu'une insertion d'un enregistrement arrivant d'un client peut modifier une case dans plusieurs endroits. Notre approche n'a pas cet inconvénient, étant sans incidence sur ces opérations. C'est la raison majeure de son choix. A notez aussi qu'il n'interfère pas avec les fonctions programmées pour la mise à jour. C'est aussi un avantage majeur si la fonction de sauvegarde doit être ajoutée au code déjà existant. C'était également notre cas.

Le fichier d'index, beaucoup plus petit est lui aussi partitionné en petites pages, les plus petites possibles. Chaque page a une taille de 512 Octets. Ces pages sont également sauvegardés que lorsqu'elles sont modifiées.

5.2.2 Calcul opérationnel de signatures algébriques pour la sauvegarde

Le traitement de la fonction fait l'objet d'un processus léger de travail (ang. working thread). La sauvegarde d'une portion d'un fichier mappé utilise la fonction *FlushViewOfFile* [M02] disponible pour ce type de fichiers. Elle ne sauvegarde que la partie (vue) modifiée de ce fichier sur le disque. On constitue aussi sur chaque serveur un tableau contenant des informations relatives à la case (carte de la case, nom de fichier, identificateur de fichier, adresse du premier serveur, clé min et clé max…). Il est mis dans le *fichier de paramètres de la case* également sauvegardé. La table est d'une grande utilité lors de l'opération de la restauration du fichier. En somme la sauvegarde donne lieu à trois fichiers images de la case sur le disque dits : le *fichier_ de données,* le *fichier_index* et le *fichier_parametres*. Le premier fichier est équivalent à la copie du fichier mappé de donnée sur disque. Le deuxième, équivalent à la copie du fichier d'index sur disque. Enfin, le fichier_parametres contiens les différents parametrtes énoncés plus haut.

Pour chaque case, la carte des signatures est mise dans une variable de type '*structure*' nommé *SIGN*. Celle-ci comporte le nom du fichier, les signatures des pages ainsi qu'un indicateur dit '*flag*', positionné lors de la première sauvegarde. Avant ce positionnement, il n'est pas utile de comparer les signatures. Le SIGN est sauvegardé sur le disque et restauré en RAM avec le fichier.

```
Struct SIGN
{
char[25]  Nom_file ;
int   signature_page[nbre_page_case][niveau_signature] ;
int   flag ;    //sauvegarde sans comparaison lors du premier stockage
}
SIGN  Signature [NbreMaxBucket]
```

Figure 5- 2 : Structure de la table de la case.

Le pseudo-code de la figure 5-3 illustre notre calcul de la signature $Sign_{\alpha,k}$ (k=1,..,*n)* pour une page de données. Le tableau *page [i]* représente le contenu de chaque page de données et 'Size', la taille de cette page. Ils constituent les paramètres d'entrée de la fonction *Signature*. Elle est calculée pour chaque page de données. La constante Size équivaut à 2^{16} dans notre cas. On utilise le type *GFElement* et un alias du type *entier* représente l'alphabet de GF (2^{16}) utilisé. On calcule les éléments ($Sign_\alpha^1$, $Sign_\alpha^2$,..., $Sign_\alpha^n$) composant la signature (n = 2 dans notre cas).

Vu les accès fréquents à la table de la case, celle ci peut résider probablement en mémoire cache L2. Dans ce contexte, une méthode basée sur la macro *Prefetch* peut être utilisée pour forcer le chargement de cette table en mémoire cache.

```
GFElement Signature (GFElement *page, int Size, int n)
        {
        GFElement returnValue = 0;
        For (int i=0; i< pageLength; i++)
        {     if (page[i] ≠ Size-1)  then
                                returnValue ^= Antilog[n*i+page[i]];  }
        return returnValue;
        }
```

Figure 5- 3 : pseudo code pour le calcul de signature algébrique d'une page de données

5.2.3

Pour les requêtes de sauvegarde, des comparaisons sont faites entre les *signatures_avant*, disponibles dans la table de la case et les *signatures_aprés*, calculées au moment de la sauvegarde par le serveur sur le fichier mappé de données. Ces comparaisons sont faites afin de détecter les pages mises à jour.

L'utilisation de petites pages permet une détection plus précise de la page ayant été mise à jour. Ceci réduit la taille des données transférées vers le disque. Par contre, cela augmente le nombre de signatures. La table des signatures sera alors plus importante. En pratique, une bonne taille de cette page varie entre 512 *Octets* et 64*Ko*. Le meilleur choix dépend essentiellement de l'application. Notre choix s'est porté sur des pages de 16Ko. Nous discutons ce choix dans le chapitre des mesures de performances.

La rapidité de calcul de nos signatures constitue un 'challenge' important. Dans ce contexte, différentes expérimentations ont été réalisées. Nous les aborderons dans le chapitre réservé aux mesures de performances. Il faut ici éviter des cas de collisions susceptibles de ne pas détecter des mises à jours éventuelles. Le cas idéal équivaut à une probabilité proche de 'zéro'. En pratique, cette probabilité est de l'ordre de magnitude d'erreurs dans le disque. Une signature algébrique constituée de $n=2$ caractères s'avère suffisante pour notre système.

L'implémentation de la table des signatures est plus efficace si elle est structurée en 'arbre' de signatures. Cela permet de retrouver plus rapidement les signatures descendantes lors de la recherche de celles qui ont été modifiées suite à une opération de mise à jour.

5.2.4 Traitement parallèle de la requête

L'organigramme de la Figure 5- 4 montre le traitement d'une requête de sauvegarde. Celle ci consiste notamment à mettre les trois fichiers énoncés plus haut sur le disque local. Cette opération est faite sur chaque serveur contenant une case pour ce fichier.

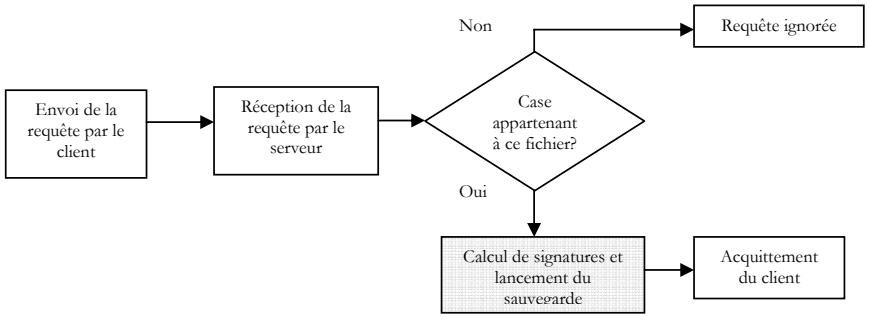


Figure 5- 4 *:* Organigramme de traitement d'une requête de sauvegarde.

5.2.5 Terminaison de la requête de sauvegarde

Cette étape intervient après la fin du traitement de la requête de stockage par les serveurs de données. Au niveau du client, un thread appelé *ReceiveResponse* est en attente sur le port UDP de la machine. Il a la

responsabilité de collecter les réponses des serveurs. Chaque réponse reçue est insérée dans un tampon jusqu'à réception de l'ensemble des acquittements. Le client effectue par la suite, la somme de ces intervalles reçus puis effectue une comparaison avec l'intervalle initial. Dans le cas ou le nouvel intervalle est plus grand, l'image du fichier au niveau du client est mise à jour. A la fin, un message informationnel est transmis à un autre thread qui se charge de les expédier vers l'application à l'origine de la requête. Il lui indique la réussite ou l'échec de l'opération de sauvegarde.

5.3 Propagation de la Sauvegarde

La propagation des mises à jour n'est pas assurée automatiquement. Elle se fait à la demande du client par le lancement d'une nouvelle requête de sauvegarde.

Des paramètres de configuration initialisés au moment de la création des fichiers mappés permettent de rendre possible cette propagation de mises à jours automatique. Ainsi, il est possible d'utiliser la commande *fssek ( )* à chaque mise à jour pour sauvegarder la page immédiatement sur disque. Elle permet de se positionner dans le fichier et de mettre à jour la page modifiée. Cette méthode présente l'inconvénient de ralentir le fonctionnement du système.

5.4 Restauration d'un Fichier à partir du Disque

Le client peut demander la restauration d'un fichier SDDS à partir du disque vers la mémoire. La Figure 5- 6 montre le déroulement de la requête de restauration [MLS03]. Pour faire parvenir cette requête au serveur, le client procède de la même manière que lors d'une requête de stockage (au code de la requête près).

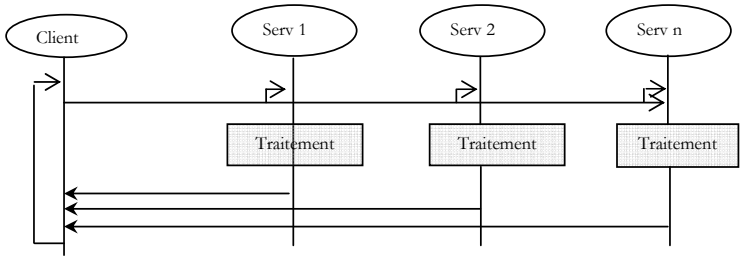


Figure 5- 5: Traitement de la Requête de Chargement.

Un serveur ne possédant pas de fichier de même nom sur son disque ignore la requête. Dans le cas contraire, la requête est traitée, puis le serveur renvoie l'acquittement au client (figure 5-5). Celui-ci agit comme pour la terminaison de la requête de sauvegarde.

Mise en forme : Puces et numéros

La restauration d'un fichier SDDS s'effectue en quatre étapes. Il concerne les trois fichiers sauvegardés lors de la requête de sauvegarde. D'abord, chaque serveur vérifie l'existence du fichier sur son disque local. Si c'est le cas, le serveur commence par le chargement des paramètres de la case stockés sur disque (Clé minimum, clé maximum…) et présents dans le *fichier_paramètres*. Il continue avec l'ouverture du fichier de données sur disque ainsi que le fichier de l'index. Puis, la création de deux fichiers mappés est faite en mémoire. La taille de ces fichiers dépend des informations contenues dans la table des paramètres déjà chargée en mémoire. On créé d'abord le fichier mappé de données puis le fichier mappé d'index. Ensuite, une copie du contenu des fichiers sur disque est faites vers les fichiers mappés correspondants. Puis, chaque serveur met à jour met à jour l'image du client en envoyant un IAM à ce dernier.

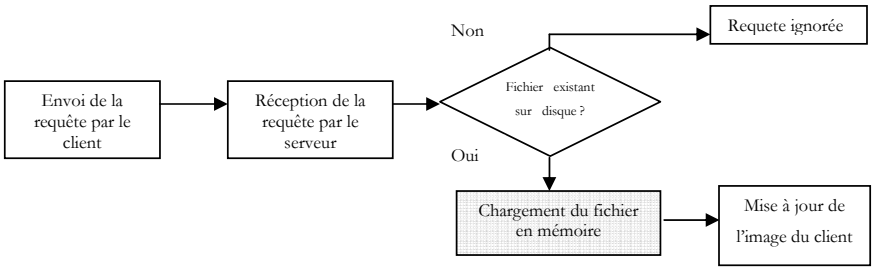


Figure 5- 6 : Organigramme de déroulement de la requête de chagement de fichier

5.5 Synthèse

Nous avons montré notre utilisation des signatures algébriques pour la sauvegarde/restauration de données dans SDDS-2005. Nous ne sauvons que les pages modifiées, sans perturber les opérations normales de mise à jour de la case. Le coût mémoire additionnel est négligeable, nos signatures étant de quatre Octets par page de 16 KO seulement. Les mesures expérimentales dans le chapitre 8 montrent que le traitement est rapide et peut procurer une accélération notable par rapport aux sauvegardes « naïves », c. à d. totales.

Nos choix de la taille de la page et de celle de la signature sont des compromis que l'analyse heuristique a montré qu'elles étaient adéquates dans notre cas. D'autres choix sont possibles et peuvent être à l'évidence plus avantageux dans d'autres contextes. D'une manière générale, les pages plus petites génèrent des écritures plus petites aussi, mais leur nombre peut augmenter. Le choix optimal pour une application peut être une décision complexe.

Divers travaux existent dans le domaine de la sauvegarde de données notamment ceux relatif au stockage avec compression de données. Cette possibilité n'est pas implémentée dans notre système. Ceci, pour des raisons de rapidité de sauvegarde et de chargement de fichiers. Néanmoins, on pourrait envisager une telle option dans l'avenir.

Chapitre

6 *Signatures Algébriques pour la Mise à Jour de Données dans SDDS-2005*

Dans ce chapitre, on discute l'implémentation d'une fonction de mise à jour (MAJ) plus efficace dans SDDS-2005. A cette fin, nous calculons les signatures algébriques au niveau de chaque enregistrement, à sa création et à chaque MAJ de la zone non-clé. Selon le vocabulaire habituel, chaque enregistrement est *estampillé* par notre signature. Ces signatures nous permettent d'une part de rendre les mises à jour concurrentes au niveau 1 (pas de MAJ perdue) [SH04, LS04]. Notre contrôle est optimiste. Ceci évite de pénaliser les performances d'accès à nos fichiers, en RAM. Dans ce contexte, nos signatures ont un avantage sur les estampilles (ang. time stamps) habituelles par le fait qu'elles sont calculées sur les clients. Notons que ces dernières sont peu commodes d'emploi pour les fichiers en RAM en général, compte tenu de la vitesse de la RAM. Ensuite, l'examen de signatures d'avant et d'après la MAJ, permet au client de n'envoyer au serveur qu'une MAJ effective. Donc, uniquement celle où l'application a effectivement changé le contenu non-clé de l'enregistrement qu'elle a passé au client pour l'écriture dans le fichier. Ce filtrage concerne aussi bien les signatures 'normales' que celles dites 'aveugles'.
Nous le montrons, expérimentalement dans chapitre 8, que l'on peut souvent réaliser une économie de communications, par rapport à l'approche « naïve » habituelle. Le gain est dû au rapport entre la vitesse

de calcul de la signature et celle d'envoi d'un enregistrement par le réseau, même rapide tel que 1 Gbs. L'économie est également due, pour la mise à jour aveugle, au gain de temps entre la transmission d'une signature seule et celle d'un enregistrement entier.

6.1 Introduction

Différents clients peuvent lire ou mettre à jour simultanément le même enregistrement SDDS. Il serait alors préférable que chaque client puisse lire l'enregistrement qu'il souhaite sans attente. Les mises à jours ne devraient pas également se chevaucher. Lors d'une opération de mise à jour de salaire, une ouverture en écriture ou l'accès simultané par deux clients pourrai engendrer un salaire erroné suite à la mise à jour par l'un ou l'autre des clients [G01, LS04]. En conséquence, on doit prévoir la sérialisabilité de ces mises à jour dans SDDS-2005. Lorsque le traitement simultané d'un ensemble d'applications est équivalent à son traitement en série, on désignera par sérialisabilité, l'exécution de ces opérations.
Il existe plusieurs approches de sérialisabilité pour gérer les interactions entre transactions concurrentes [G01, LY81]. Parmi ces approches, on cite le verrouillage, l'estampillage, la date de valeur et les Signatures. Dans l'approche par estampillage, on donne un ordre total sur les transactions et on vérifie que les attentes entre transactions sont conformes à cet ordre. L'approche par verrouillage est la solution la plus implantée. C'est une technique de prévention de conflits basée sur le blocage des objets par des verrous en lecture ou écriture avant d'effectuer une opération de sélection ou de mise à jour. Le protocole le plus classique est le *verrouillage à deux phases*. Dans ce protocole, toutes les acquisitions de verrous sont faites avant la première libération. On dit qu'il y a une phase d'expansion suivie d'une phase de libération. Le protocole de *verrouillage à deux phases* garantit la propriété de sérialisabilité mais ne permet pas d'éviter les situations d'interblocage. Un interblocage se produit lorsque une transaction est bloquée en attente de ressources provenant d'une autre transaction. Ceci n'est pas le cas dans SDDS-20005.

6.2 Traitement de Requêtes

A la création d'un enregistrement, par la requête d'insertion, le client calcule la signature algébrique de la zone non-clé. Le calcul produit la signature algébrique sur 4 Octets, consistant de 2 symboles dans GF (2^{16}). Sa précision (l'inverse de la probabilité de collision) déjà discutée dans le Chapitre précédent, est suffisante pour assurer en pratique à toute zone non-clé une signature unique. Le client ajoute cette signature à l'enregistrement et l'adresse à un serveur, choisi selon la clé. Le serveur correct vérifie que l'enregistrement reçu n'est pas dans la case et l'insère dans ce cas.

Par la suite l'enregistrement peut subir une mise à jour. Celle-ci peut être *normale* ou *aveugle*. Les mises à jour peuvent être demandées en concurrence par plusieurs clients, de la part de leurs applications locales. On détaille plus loin ces deux types de mise à jour.

6.2.1 Gestion de la concurrence par Utilisation des signatures algébriques

Pour une mise à jour plus efficace dans SDDS-2005, on opte pour l'approche utilisant *les signatures algébriques* [ML06]. Ce choix est justifié pour plusieurs raisons. La plus significative étant que ces signatures permettent une exécution très flexible du module client. L'objectif général est de rendre invisible aux clients cette gestion de concurrence. Dans notre approche, ces derniers ne sont jamais en état d'attente contrairement aux autres démarches de résolution des conflits concurrentiels. Les données ne sont pas verrouillées lors de la transaction, on évite alors les situations d'interblocage. De plus, l'utilisation de ces signatures permet d'éviter l'envoi d'enregistrements quand les valeurs de ceux ci ne sont pas changées réellement. Ceci est aussi valable pour une caméra de surveillance nécessitant la sauvegarde de grande quantité de données, inutile dans le cas de non modification des images collectées. De plus, Cette approche ne nécessite pas un gestionnaire de transaction au niveau des applications. Notre démarche est inspirée de l'approche optimiste concernant notamment le contrôle de concurrence dans *Ms Access*. Cette approche n'est pourtant pas l'option traditionnelle répandue dans la littérature. Nous utilisons le principe d'*image avant* et d'*image après* pour détecter les similarités et éviter le transfert inutile de données. Ici, les images avant et après correspondent aux signatures algébriques avant et après une mise à jour [LS04].

Considérons R_b l'*image avant* d'un enregistrement et S_b la signature correspondante. L'*image avant* correspond au contenu de l'enregistrement R concerné par la mise à jour. Apres une opération de mise à jour, l'*image après* sera R_a et sa signature S_a. Si R_a dépend de R_b comme le cas d'une mise à jour sur un salaire (*Salaire :=Salaire+0.01*Salaire*), on dit que la mise à jour est *Normale*. Dans l'autre cas, une mise à jour est dite *Aveugle* dans le cas où R_a est indépendant de R_b. Ce dernier cas correspond est valable pour l'affectation *Salaire :=1000* ou dans le cas d'une sauvegarde de l'image d'une vidéo surveillance. L'application aura besoin de R_b dans le premier cas. Ceci n'est pas le cas lors d'une mise à jour aveugle.

Lors d'une mise à jour normale dans SDDS-2005, le client lance une requête de mise à jour après confirmation du serveur de l'existence de l'article à mettre à jour. Ceci n'est pas le cas pour la mise à jour aveugle. Dans celle ci, il s'agit de mettre à jour un article sans qu'on soit certain de son existence.

6.2.2 Calcul opérationnel de signature pour un enregistrement

Un processus léger de travail (thread) s'occupe du calcul de signature pour chaque enregistrement. Suivant le type de MAJ, le calcul est fait soit au niveau du serveur ou du client. Comme énoncé dans le chapitre 4, suivant les proprités d'une signature algébrique, un enregistrement ne doit pas dépasser 64 KOctets. Par ailleurs, la taille de celui ci est variable. Lors d'une mise à jour d'un enregistrement, le thread en question procède d'abord à la comparaison des enregistrements par la taille. Ce n'est qu'en cas d'égalité que le serveur calcule puis compare les signatures.

Le pseudo-code de la Figure 6- 1 illustre notre calcul de la signature $Sign_{\alpha,k}$ (k=1, 2*)* pour un enregistrement. Le tableau *record [i]* représente le contenu de cet enregistrement et 'Size', sa taille.

```
GFElement Signature (GFElement *record, int Size, int n)
        {
        GFElement returnValue = 0;
        For (int i=0; i< RecordLength; i++)
        {     if (record[i] ≠ Size-1)  then
                                returnValue ^= Antilog[n*i+record[i]];  }
        return returnValue;
        }
```

Figure 6- 1 : Calcul de signature pour un enregistrement.

6.2.3 La mise à jour normale

Dans SDDS-2005, la mise à jour normale d'un enregistrement *R* se présente à l'issue du succès d'une requête de recherche de cet enregistrement. Nous nous basons sur le concept de signature avant et signature après pour chaque enregistrement.

L'application lance d'abord une recherche concernant l'enregistrement R qu'elle veut mettre à jour. L'application transmet la nouvelle valeur de R notée R_a (R after changement) au client. Celui-ci envoi la clé de l'enregistrement à mettre à jour au serveur (requête de recherche). A ce niveau, le serveur fait une *recherche par clé* et lorsque l'enregistrement est trouvé R_b, il l'envoie au client.

Le client calcule la *signature après* S_a sur la nouvelle donnée et la *signature avant* S_b sur l'enregistrement R_b qu'il a éventuellement reçu. Les deux signatures S_a et S_b sont comparées au niveau du client. Deux cas se présentent lors de cette comparaison:

- La *signature avant* est égale à la *signature-après* ($S_b = S_a$), on est dans le type de mise à jour normale sans changement, appelée aussi '*pseudo mise à jour*'. Le client renvoi un message informationnel à l'application. L'opération de mise à jour se termine.

- La *signature avant* est différente de la *signature-après* ($S_b \neq S_a$). Dans ce cas, on est dans le type de mise à jour avec changement (*mise à jour effective*). Le client envoie, alors, la s*ignature avant* S_b avec la nouvelle donnée R_a au serveur. A ce niveau, le serveur recalcule la *signature serveur* S_s (signature du contenu de l'enregistrement à ce moment) et la compare avec la signature avant. Deux cas se présentent alors :

    - La *signature serveur* S_s est égale à la *signature avant* S_b ($S_s = S_b$). Dans ce cas, la donnée n'a pas été changée par un autre client ce qui autorise la mise à jour. L'enregistrement R est mis à jour avec la nouvelle valeur reçue Ra. Le serveur envois un *Ack* positif au client.

    - La *signature serveu*r S_s est différente de la *signature avant* S_b ($S_s \neq S_b$). Dans ce cas, la donnée a été changée par un autre client ce qui n'autorise pas la mise à jour. Cela veut dire qu'une mise à jour concurrente est intervenue entre le temps de lecture de l'enregistrement R_b et la réception du nouvel enregistrement *Ra.* Le serveur envois un *Ack* négatif au client.

En plus de la propriété de comparer directement les signatures plutôt que des enregistrements, nos signatures permettent également d'éviter des transferts inutiles de grands enregistrements si leurs valeurs demeurent inchangées après une opération de mise à jour. On l'appellera dans ce cas une p*seudo mise à jour.*

6.2.3.1. Déroulement de la requête

Le client envoi la requête *UpdateRequest* à l'ensemble des serveurs en utilisant un message multicast (Figure 6- 2*)*. Les serveurs ne contenant pas de cases de même nom que celui spécifié dans les paramètres de la requête, ignorent la requête. Dans le cas où le serveur contient dans son intervalle la clé de l 'enregistrement, il traite la requête en lançant le processus de mise à jour normale.

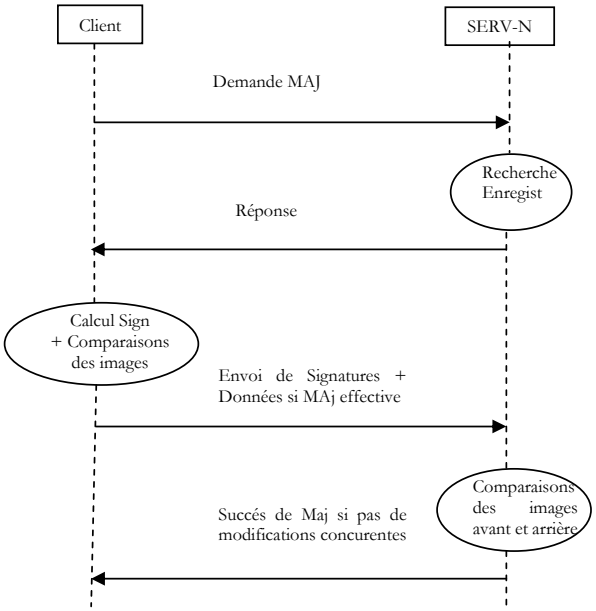


Figure 6- 2 : Déroulement d'une Requete de Mise à Jour Normale

6.2.4 La mise à jour Aveugle

L'application lance la requête de mise à jour aveugle [LS04] en transmettant la clé et la nouvelle donnée R_a au client. Celui-ci, comme pour la mise à jour normale, calcule d'abord la *signature après* S_a sur la nouvelle donnée et transmet la clé au serveur. Celui ci calcule la *signature avant* S_b de cet enregistrement (s'il existe) puis l'envois au client. A partir de ce moment, le client et le serveur procèdent de la même façon que lors d'une mise à jour normale.

L'envoi de la *signature avant* S_b seule permet d'éviter le transfert inutile de l'enregistrement R_b vers le client (pas de recherche). Ceci est souvent le cas lorsqu'on veut éviter le transfert d'enregistrements de grandes tailles (cas des images d'une caméra de surveillance par exemple) comme c'est le cas dans la plupart des DBMS (Data Base Manager System) [V93].

6.2.4.1. Exemple d'une mise à jour aveugle

Prenons le cas d'un client qui veut mettre à jour deux enregistrements se trouvant sur deux serveurs de données différents. L'enregistrement R_1 se trouvant au serveur S_1 contient initialement la valeur v1. Cet enregistrement est mis à jour avec la même valeur *v1*. L'enregistrement R_2 se trouvant au serveur S_2,

contient initialement la valeur *v2* et il est mis à jour par le client avec la valeur *v'2* différente de *v2* (Figure 6- 3).

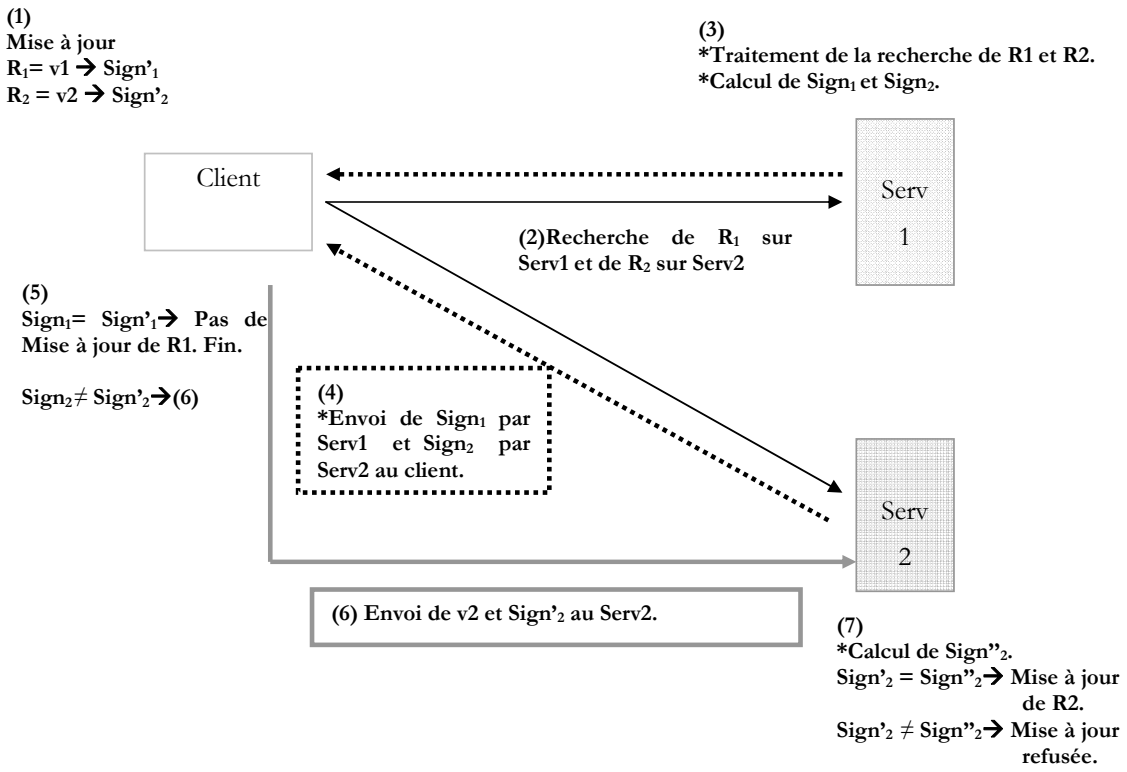


Figure 6- 3: Exemple de Mise à Jour Aveugle.

Le client commence par le calcul des signatures algébriques de *v1* et *v2* ayant pour valeur Sign'$_1$ et Sign'$_2$ respectivement (l'image après des enregistrements R_1 et R_2 resp). Ensuite, il envoi des requêtes de recherche de R_1 et R_2 aux serveurs de données respectifs. Au niveau de chaque serveur de données, une recherche de l'enregistrement demandé (R_1 au niveau de S_1 et R_2 au niveau de S_2) est effectuée. Puis, chaque serveur calcule la valeur de l'image avant pour l'enregistrement demandé. Supposons que ces valeurs sont $Sign_1$ pour R_1 et $Sign_2$ pour R_2. Ensuite, chaque serveur envois au client la valeur calculée. Celui ci compare la valeur de la signature reçue avec celle déjà calculée auparavant. A la suite de cette comparaison, Sign$_1$ et Sign'$_1$ sont égaux. Le client ne procédera pas alors à la mise à jour de R_1 vu que sa valeur est inchangée.

Dans le cas de R_2, la signature $Sign_2$ est différente de *Sign'2*. Le client envois la nouvelle valeur *v2* de R_2 et la signature $Sign'_2$ au serveur S_2. Ce dernier recalcule la signature de l'enregistrement $Sign''_2$. Celle ci est comparée à la signature reçue. S'il y'a égalité, la mise à jour est autorisée et l'enregistrement R_2 est mis à jour avec la valeur v2. Dans le cas contraire, la mise à jour est refusée car entre temps, un client concurrent a modifié le contenu de R_2.

6.2.5 Autre approche : Signature sauvegardée

Dans cette approche, aucun calcul de signatures ne se fait au niveau des serveurs de données. Pour accélérer notre processus, nous proposons d'introduire un nouveau champ dans l'en tete d'un enregistrement (figure 3-5). Celui ci comporte la signature d'un enregistrement calculée au moment de son insertion. Le serveur a la possibilité de 'lire' directement cette signature sans avoir à recalculer sa valeur à chaque fois d'autant plus que la signature est sur 2 Octets. Dans cette approche, un minimum de calcul de signatures est effectué du fait que cela est déjà fait à l'avance. Ces signatures seront sauvegardées au sein de chaque enregistrement.
Ce champ ajouté à la structure de l'en tête d'un enregistrement sert également lors d'une recherche par le contenu (recherche complète). On détaille cela dans le chapitre 6.

6.3 Synthèse

Nous avons montré l'utilisation des signatures algébriques pour une mise à jour efficace dans SDDS-2005. Théoriquement, Ceci procure un gain de temps important économies lors de la communication client-serveur d'une SDDS ainsi qu'un traitement plus rapiide d'autant plus que l'utilisation de signatures algébriques permet aux clients d'accéder aux données sans attente ni interblocage. Un schéma de contrôle de concurrence optimiste est implémenté, utilisant les signatures comme estampilles. On montre l'utilité de l'approche pour les fichiers en RAM tout particulièrement, par le fait que ces estampilles ne sont calculées que par les clients. L'utilisation de concepts de *signature avant* et *signature après* permet de n'envoyer la nouvelle valeur de la donnée que dans le cas ou celle ci est réellement modifiée.

Chapitre

7 Signatures Algébriques Cumulatives

Dans ce chapitre, nous présentons une extension des signatures algébriques, les signatures cumulatives [LMS05a, LMS05b]. Elles encodent chaque caractère par sa signature cumulative. Ces signatures sont calculées sur le client à l'insertion ou lors de mise à jour d'un enregistrement, d'où leurs appellation 'signatures pré calculées'. Cette manière d'encoder le contenu des enregistrements procure une protection contre la vue accidentelle de données par les administrateurs de serveurs. Ces signatures possèdent aussi des propriétés utiles pour différents types de recherche de chaînes de caractères. Celles-ci peuvent s'effectuer en parallèle sur les serveurs, sans décodage du contenu des enregistrements. Pour certaines recherches, il est inutile que le client envoie aux serveurs la donnée à rechercher. Envoyer sa signature algébrique suffit. Ceci peut être fort intéressant pour des chaînes longues, à l'évidence.

Il apparaît par ailleurs que les algorithmes de recherche par signatures cumulatives s'avèrent très efficaces. Cette efficacité peut justifier leur utilisation même si la protection n'est pas le but premier. La technique peut alors être vue comme un apport général au domaine de la recherche de chaînes. Elle se situe dans ce cadre dans la classe des algorithmes à pré-calcul (prétraitement, selon un autre vocabulaire courant) de données à explorer. Cette classe est utile à toute application dans laquelle on écrit une donnée une fois pour la rechercher de multiples fois. Il s'agit notamment de bases de données, *ipso facto*.

Dans ce qui suit, nous introduisons d'abord le concept de la signature cumulative. Puis, on définit l'algorithme d'encodage et de décodage. Ensuite, on aborde la recherche des enregistrements dans SDDS-2005 par les chaînes de caractères dans les zones non-clé. . On détaille les fonctions de recherche de chaînes disponibles dans SDDS-2005. On présente notamment l'approche choisie à la résolution de collision. Puis, pour mieux situer les fonctions proposées par rapport aux techniques alternatives de recherche de chaînes, on discute notamment l'algorithme de Karp-Rabin et la recherche par la méthode XOR et les signatures algébriques dans GF (2^{16}). On compare les performances respectives dans le Chapitre 8.

7.1 Signatures Algébriques Cumulatives dans SDDS-2005

7.1.1 Protection Contre les Vues Accidentelles

Dans SDDS-2004, la partie non clé d'un enregistrement contenait les données originales insérées par l'utilisateur. L'administrateur d'un serveur SDDS pouvait découvrir ces données, même par accident. Par exemple, lors d'une opération de maintenance utilisant le « dump », les champs de mémoire peuvent apparaître en chaînes ASCII lisibles « à l'œil nu ». L'utilisateur d'une SDDS, notamment dans le cadre P2P, pourrait ne pas apprécier cette perspective. L'administrateur peut-être non plus, d'ailleurs. On pense dès lors à un encodage comme fonction standard avec un décodage systématique en retour. Cependant, l'utilisation efficace d'une SDDS présente la contrainte d'un scan efficace. Ce qui implique une exploration des enregistrements sur les serveurs. La fonction primaire d'un encodage typique est précisément d'éliminer une telle possibilité.

Un certain compromis est dès lors nécessaire. On peut s'inspirer à cette fin de la vie courante. La plupart de documents dans des organisations sont simplement envoyés sous une enveloppe papier. Celle-ci n'est une protection que contre la découverte *accidentelle* des informations dans ces documents. Une protection plus forte, par une boîte blindée ou un courrier sécurisé est possible. Elle est extrêmement rare. Normalement, la connaissance des conséquences pénales de la violation du secret de la correspondance suffit.

Le concept de la *protection contre la vue accidentelle* et celui de signatures cumulatives en tant qu'un outil dans ce but suivent cette analogie. Les explorations de données locales à un serveur visées par cet outil sont les diverses algorithmes de recherche de chaînes de caractères. De telles recherches sont parmi les plus utilisées et se situent aussi parmi les techniques fondamentales de l'informatique. Le volet original des signatures cumulatives est qu'elles permettent de faire ces recherches sans décodage local de

données. Comme nous avons mentionné, les algorithmes de recherche résultant s'avèrent très efficaces. Au point de justifier leur utilisation autonome, en dehors de la préoccupation sécuritaire.

7.1.2 Définition d'une Signature Cumulative

Considérons un enregistrement comme une suite d'éléments de corps de Galois *GF* (2^f). Chaque élément est composé de f bits. Ces éléments sont les caractères ASCII ($f = 8$) ou Unicode ($f=16$). Notons p_i le $i^{ème}$ symbole de la zone non-clé de l'enregistrement P contenant l symboles $(p_0, p_1,...p_{l-1})$. Notons α un élément primitif de GF (2^f), par exemple $\alpha = 2$ pour nos deux GFs. La s*ignature algébrique cumulative* de p_k s'obtient en calculant la signature algébrique de la chaîne $p_0, p_1,...p_k$ [LMS05a, LMS05b]. Nous considérons ici uniquement les signatures cumulatives à 1 symbole. Une telle signature se calcule donc pour chaque p_k comme suit :

$$Sign\ (p_k) = \sum\ p_i\ \alpha^i \qquad i = 0, 1,..,k$$

La probabilité de collision de la signature cumulative est en général ainsi de 2^{-f}. Pour la multiplication dans le cadre de calcul de signatures cumulatives, on continue d'utiliser nos tables *Log/Antilog*, à double taille (2^{2f}-2) pour cette dernière, pour éviter l'utilisation du modulo 2^f-1. Cependant, on montrera que l'utilisation du calcul modulo nous est alors nécessaire dans le cas de la recherche d'une chaîne dans une zone dépassant 255 caractères.

7.1.3 Encodage de données

L'encodage concerne chaque enregistrement dans SDDS-2005. La valeur d'un caractère encodé est obtenue en calculant sa signature cumulative. Dans une opération d'encodage, chaque élément individuel p est remplacé par un autre symbole noté p'' ci-dessous tels que la valeur de p'' est la signature du préfixe ayant pour le dernier élément le caractère p. Cette signature encode non seulement l'élément p mais tous les caractères précédant cet élément. L'avantage est tel qu'une simple comparaison de 2 caractères, le caractère p et celui de la chaîne à comparer, permettra de décider si tous les éléments qui les précèdent sont égaux. Une condition doit être néanmoins vérifiée, les 2 chaînes doivent être de même taille.

Nous notons P_i le préfixe de P jusqu'à la position p_i. Au moment de l'encodage de chaque caractère de P $(p_1, p_2, .., p_n)$, le client calcule la signature algébrique pour chaque p_i obtenant p''_i. Le calcul basique serait de calculer chaque signature entièrement. Ceci rendrait le calcul d'encodage lourd, surtout pour des chaînes longues. On peut observer par contre que toute signature algébrique a la propriété (algébrique) suivante :

$$p''_i = p''_{i-1} + p_i \alpha^i ; \; i > 0.$$

Ce qui veut dire que le calcul de la signature suivante peut réutiliser entièrement le résultat précédent. Ce qui donne à l'encodage la complexité linéaire de $O(l)$. Le calcul dans SDDS-2005 se fait dès lors de la manière suivante.

Pour chaque $i = 0,1 \ldots l-1$, Le client calcule d'abord le produit $p'_i = p_i \alpha^i$. Le calcul opérationnel pour $l < 2^f$ est en fait :

$$p'_i = Antilog (\log p_i + i)$$

Pour une zone non-clé plus longue que 2^f-1 ce qui équivaut en pratique aux chaînes de caractères de taille supérieure à 255 Octets, le calcul est alors :

$$p'_i = Antilog ((\log p_i + (i \mod 2^f - 1))$$

Ensuite le client calcule la somme :

$$p''_i = p''_{i-1} \; XOR \; p'_i$$

Notre méthode d'encodage (Figure 7- 1) est ainsi fort peu coûteuse en traitement. On a, par symbole, seulement 1 accès par table utilisée (*Antilog* et *Log*) et un 'XOR'. Plus le calcul modulo éventuellement. Dans le chapitre relatif aux mesures de performances, on analyse le coût de cet encodage en détail en le comparant notamment à celui de l'opération d'insertion sans encodage. On montre qu'il s'agit d'un surcoût négligeable.

Dans SDDS-2005, l'encodage concerne la partie non clé de chaque enregistrement. Néanmoins, la partie clé (en tête) peut être aussi concernée. L'adressage se fera alors suivant la valeur originale de la clé ce qui nécessite un traitement supplémentaire lors de l'adressage au niveau des serveurs de données. Le décodage devient nécessaire pour progresser au sein des indexes internes. Ceci n'est pas le cas dans SDDS-2005.

Malgré son calcul simple et rapide, notre encodage suffit pour la protection contre une vue accidentelle. La tentative de décodage ne peut être que frauduleuse et doit être assez complexe d'ailleurs pour un amateur. D'abord la valeur de α n'est pas stockée sur le serveur, pendant que nos GFs ont de nombreux éléments primitifs. Ensuite la valeur d'un même caractère est en général encodée différemment non seulement selon sa position dans la chaîne, mais aussi et surtout selon le contenu qui le précède. Donc ni un utilitaire de dump postmortem, ni un examen visuel de propriétés apparentes de valeurs binaires, hexa, ASCII… visibles ne peuvent en général visualiser ni même donner l'idée de valeurs réelles de données stockées.

7.1.4 Décodage de données

Suite à une opération de recherche et après traitement de celle ci par les serveurs de données, les enregistrements sélectionnés sont envoyés au client sous leur forme encodée. Ceci, d'abord pour des raisons de sécurité expliquées, puis pour celles de vitesse de traitement sur le serveur. Le décodage est fait par le client. Il ne concerne à présent que la partie non clé, comme l'encodage.

Pour décoder l'enregistrement encodé *P"* reçu, il faut trouver tout p_i. Le client effectue alors, pour chaque caractère le calcul suivant. On « profite » de nouveau d'une propriété algébrique de toute signature algébrique qui est en l'occurrence comme suit :

$$p'_i = p"_i - p"_{i-1},$$

où $p'_i = p_i \alpha^i$, on le rappelle. La soustraction dans les GFs utilisé étant calculée par XOR, le client fait le calcul suivant :

$$p'_i = p"_i \text{ XOR } p"_{i-1}$$

Reste à faire la division par α^i (chaque élément non nul dans GF a son inverse). Ce calcul se fait pour une zone plus courte que *l* élément comme :

$$p_i = Antilog\ (log\ p'_i - i)$$

Autrement, on calcule la valeur de *i* modulo $2^f - 1$ comme pour l'encodage.

On voit qu'avec notre méthode, deux accès aux tables *Log* et *Antilog*, un *XOR* et peut-être un calcul modulo suffisent par symbole. Le décodage est ainsi aussi rapide que l'encodage. La complexité de l'opération est de *O (l)*.

Exemple 7.1

Considérons « *UNIVERSITE_DAUPHINE* » est la partie non clé d'un enregistrement P. Soit $\alpha = 2$ l'élément primitif dans GF (2^8). Au niveau du client, le premier calcul produit *P'*:

$$P' = (p'_1, p'_2, \ldots, p'_{19}) = (2U, 2^2N, \ldots, 2^{19}E)$$

Comme l'addition et la soustraction, la multiplication et la puissance sont aussi calculées dans GF. Les caractères *U, N, I*…sont représentés par leurs codes ASCII.

L'encodage du 5ème élément '*E*' est calculé suivant la formule

$$p'_5 = Antilog\ (\log E + 1).$$

Pour l'encodage du 10ème élément, aussi '*E*' on a :

$$p'_{10} = Antilog\ (\log E + 10).$$

Remarquons ici que $p'_{10} \neq p'_5$

Ainsi, Le calcul de la signature cumulative p"$_{10}$ sera calculée comme suit $p"_{10} = p"_9 + p_{10}\, 2^{10}$

$P" = (p"_1, p"_2, \ldots, p"_{19}) = (2U, p"_1 + 2^2N, \ldots, p"_{18} + 2^{19}E)$

Le décodage de *P"* sur le client donne :

$$p_5 = Antilog\ (\text{Log}\ E + 1 - 1) = E \text{ et } p_{10} = Antilog\ (\text{Log}\ E + 10 - 10) = E.$$

La signature partielle (l symboles) pré calculée est alors $Sign\ (\text{P}) = 2U + 2^2N + \ldots + 2^{19}E$.

Rappelons que le (+) correspond au *XOR* dans GF (2^8).

Notons que *Sign (P)* est équivalente à $p"_{19}$ (signature cumulative pré calculée). Dans notre exemple, le changement du ' ' en '_' dans l'enregistrement entraînera une modification de *Sign* (*P*). Aussi, une erreur de frappe '*EN*' au lieu de '*NE*' entraînera une erreur avec une probabilité de 1- $1/2^8$=0.996 et l'utilisation d'une signature à 2 symboles augmentera encore plus cette probabilité.

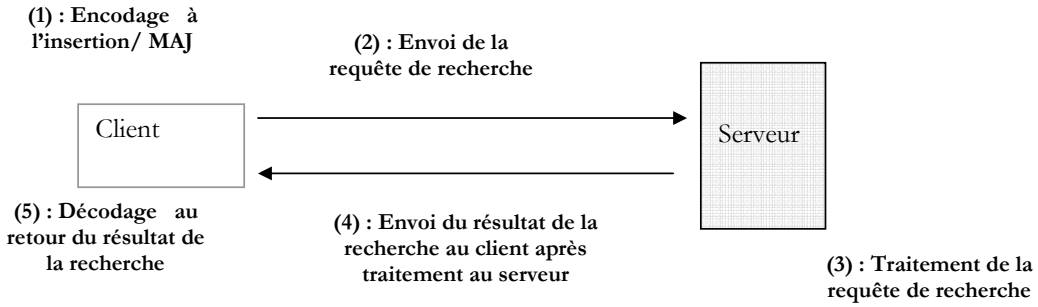


Figure 7- 1: Processus d'Encodage / Décodage.

Par ailleurs, cet encodage est complètement transparent pour les serveurs de données. L'encodage ainsi que l'opération inverse, le décodage, sont réalisées au niveau du client. Le serveur reçoit les données et les traites dans cet état. En effet, pour accéder aux enregistrements, il n'est pas nécessaire de décoder les données au niveau des serveurs de données. Ceci est le cas aussi bien pour SDDS LH* que pour les SDDS basé sur RP*. De plus, on verra dans la section suivante que l'encodage / décodage ne coûte pas grand chose du point de vue temps de traitement d'autant plus que les signatures cumulatives se révèlent très utiles pour les différents types de recherche qu'on aborde dans la suite de ce chapitre.

7.2 Recherche de Chaînes de Caractères dans SDDS-2005

Dans les versions précédentes du prototype, on ne pouvait rechercher un enregistrement que par sa clé. On ne pouvait pas demander de recherches dans la zone non-clé. Notre système offre plusieurs

fonctions de recherche correspondantes. Il s'agit de recherches par de chaînes de caractères dans cette zone. Les recherches sont faites en parallèle sur les serveurs. On explore en général les signatures cumulatives sans décodage. Sauf pour la fonction dite de recherche complète qui explore la signature algébrique de toute la zone non-clé. Nous présentons ci-dessous les fonctions offertes par SDDS-2005 à présent. Il s'agit successivement de recherches suivantes.

- Recherche par préfixe. On retourne tous les enregistrements qui ont le préfixe donné par l'application.
- Recherche complète. On retourne les enregistrement dont la totalité de la zone non clé contiens la chaîne donné par l'application.
- Recherche par préfixe. On retourne tous les enregistrements qui ont le préfixe donné par l'application.
- Recherche par une chaîne de caractères. On retourne tous les enregistrements contenant la chaîne recherchée par l'application dans la zone non-clé. On offre deux fonctions de recherche dites respectivement *séquentielle* et par *n-Grammes*. Les deux opèrent sur les enregistrements encodés (en signatures cumulatives).
- Recherche du plus grand préfixe. On retourne l'enregistrement contenant le préfixe le plus long, commun au préfixe donné par l'application.
- Recherche de la plus grande chaîne commune. On retourne les enregistrements ayant la chaîne commune la plus longue avec la chaîne donnée par l'application.

Comme méthode alternative, on étudie la recherche par la méthode XOR qui s'appuie sur le calcul de signatures algébriques dans GF (2^{16}). Cette fonction n'est pas intégrée à présent, mais apparaîssait également potentiellement utile. Nous l'utilisons dans le Chapitre 8 pour l'analyse expérimentale comparative de performances.

7.2.1 Recherche par Préfixe

Cette fonction permet à l'application de rechercher dans un fichier SDDS-2005, tout enregistrement La recherche d'un préfixe est parmi les plus importants types de recherches permises dans notre prototype SDDS-2005. Supposons qu'un client demande la recherche d'un préfixe à *l* caractères dans les enregistrements d'une case de données.

La démarche standard consiste à parcourir séquentiellement la partie non clé de l'enregistrement dans le cas d'égalité entre le premier caractère, le deuxième et de même pour les caractères suivants. Notre

stratégie consiste à ce que le client calcule la signature de ces l caractères puis envoie au serveur cette signature ainsi que la taille l du préfixe recherché. Dans le cas de caractères *ASCII*, la signature est dans $GF(2^8)$. Elle est dans $GF(2^{16})$ pour les caractères *UNICODE*.

En recevant la requête, le serveur se met à parcourir séquentiellement tous les enregistrements de la case, en ordre de clés. Pour chaque enregistrement P, il lit le symbole p''_l de la partie non clé, qui constitue la signature algébrique du préfixe de cette zone. Il compare ensuite p''_l à la signature reçue. Si le résultat est négatif, il passe à l'enregistrement suivant. Autrement il qualifie l'enregistrement pour le test ultime de la collision. Dans ce type de recherche, la résolution de collision est faite au niveau du client. Pour ce type de recherche, le serveur ne possède que la signature du préfixe à rechercher. On aborde la technique de résolution de collision plus en détail dans la section 7.4.

Analyse

On voit que le calcul ci-dessus nécessite une seule comparaison pour éliminer ou qualifier un enregistrement. Une fois sur 2^f, on peut avoir besoin de vérifier pour rien la collision. Ce qui peut coûter dans le cas le pire $O(l)$ comparaisons. Les collisions devraient être rares en pratique, vers une par 2^f enregistrements visités. Puis elles sont effectuées sur le client. La résolution de collision est abordée dans la section 7.4. En supposant qu'il n'y a que peu d'enregistrements à sélectionner par rapport à la taille de la case, la complexité globale par l'enregistrement de notre recherche par préfixe devrait être $O(1 + \varepsilon)$. Celle de la recherche sur le serveur est $O(1)$. Cette borne est à l'évidence le meilleur résultat possible.

Il faut rappeler que la recherche basique ou même tout algorithme alternatif, peut avoir bien plus souvent à examiner plusieurs symboles de l'enregistrement. Notamment dans les cas relativement courant ou beaucoup d'enregistrements commencent par la même chaîne, *http://www.* par exemple. Ici, l'algorithme basique aurait à faire au moins onze comparaisons par enregistrement, étant donc au moins onze fois plus lent que le notre. La supériorité de notre approche vient du fait que le symbole prélevé contient en fait l'info non seulement sur sa propre valeur, mais également celle sur tous ce qui le précède. La signature cumulative agit sur chaque symbole comme une fonction de hachage sophistiquée, avec les propriétés intéressantes d'une signature en général. Telles que la signature de 'XY' est en général différente de celle de 'YX' pour ne rappeler qu'une seule de ces propriétés (Chapitre 4)

Exemple 7.2

Considérons deux chaînes de caractères P_1=« *CHINE* » et P_2=« *CHIEN* ».

Le calcul par des opérations 'XOR' successives des symboles de chaînes P_1 et P_2 ne permettent pas de détecter une différence entre celles ci tandis que l'utilisation des signatures cumulatives permet de faire la distinction entre ces deux chaînes Leurs valeurs ne seront pas les mêmes.

De plus, par souci de protection de nos données contre les vues accidentelles, le décodage n'est fait qu'au niveau du client et s'avère utile au niveau des serveurs de données sans que ceux ci ne s'occupent d'un quelconque décodage.

Exemple 7.3

Considérons un enregistrement (Figure 7- 2) dont la partie non clé (non encodée) contient la chaîne de caractère P: « *Université Paris Dauphine* ».

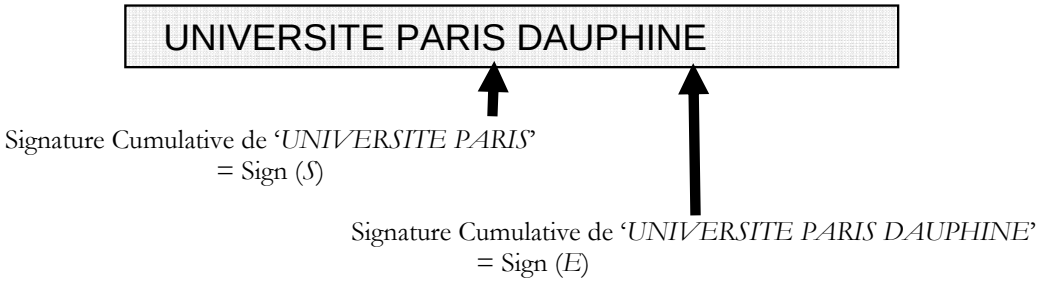


Figure 7- 2: Exemple de Recherche du Préfixe

La signature cumulative de « *Université Paris* » est obtenue en prélevant la signature de « *s* ». La signature cumulative de « *Université Paris dauphine* » est équivalente à la signature du dernier '*E*' dans l'enregistrement. Supposons que le client recherche la chaîne « *Université* » dans *P*. Cette recherche est satisfaite en une seule opération. Le serveur procède à la lecture de la signature du $10^{ème}$ caractère p_{10}. Il compare la signature reçue à p_{10}. Dans ce cas, ces deux caractères sont égaux. Une vérification de collision est faite au niveau du client. S'il n'y a pas de collision, l'enregistrement *P* est sélectionné cette recherche.

Une recherche de la chaîne « *Université Paris 9* » dans le même enregistrement ne donnera pas le même résultat. En effet, suivant la taille de la chaîne à rechercher, le serveur comparera la signature reçue à la valeur du caractère '*P*'.

7.2.2 Recherche complète

La recherche *complète* consiste à rechercher tout enregistrement dont la zone non-clé en entière est égale à la chaîne donnée par l'application. A l'évidence, il s'agit du cas limite de la recherche par préfixe. Si *n*

est la longueur de la zone non-clé, alors il suffit de comparer d'abord si $l = n$. Si c'est le cas, alors on compare p''_n à la signature reçue du client. La recherche complète peut donc être réalisé par une modification triviale de la fonction de recherche par préfixe.

Toutefois, on dispose dans tout enregistrement de SDDS-2005 de la signature de l'enregistrement calculée pour les MAJs. A chaque insertion, le client pré calcule la signature algébrique (sur 2 symboles) de chaque enregistrement et l'insère au sein de la structure de l'en-tête de celui ci (Figure 3- 5). Ce pré calcul est fait lors de l'insertion ou la mise à jour de cet enregistrement au niveau du client. Nous rappelons que cette signature comporte 2 symboles de 2 octets. Sa probabilité de collision est donc de l'ordre de 2^{-32}. A comparer à 2^{-16} ou 2^{-8} pour les signatures cumulatives utilisés dans SDDS-2005. La différence permet d'éviter le test de la collision sur le client, accélérant le traitement. C'est cette dernière approche qui a donc été adoptée pour la recherche complète sous SDDS-2005.

La complexité de l'opération en nombre de comparaison sur le serveur par enregistrement est de 0 (1), comme pour la recherche par préfixe Une seule comparaison est alors faite entre la signature reçue du client (supposons S) et celle (Sg) se trouvant dans l'en-tête de l'enregistrement. Cela se fait par le parcourt de la zone clé (en tête) de chaque enregistrement.

7.2.3 Recherche de Chaînes

Comme déja mentionné, SDDS-2005 offre deux types de recherche de chaînes (ang. string search, pattern matching). La recherche dite *séquentielle* permet au client de n'envoyer que la longueur et la signature à 1 symbole de la chaîne recherchée. Elle est mieux adaptée aux chaînes relativement longues par rapport aux enregistrements visités. Alternativement, il y a la recherche dite par *n-Grammes*. Le client envoie alors la chaîne recherche encodée. Il y a ensuite un prétraitement de la chaîne reçue sur chaque serveur, inconnu de la méthode séquentielle. La méthode est néanmoins plus avantageuse que la séquentielle pour les enregistrements longs et les chaînes relativement longues, mais plusieurs fois moins longues que les enregistrements visités. Elle est alors en général plusieurs fois plus rapide que celle séquentielle.

7.2.3.1. Recherche Séquentielle

Le client commence par calculer la signature de la chaîne reçue de l'application. Cette signature est à 1 symbole, sur 8 ou 16 bits selon le code caractère utilisé. Le client diffuse la signature calculée S et la longueur de la chaîne l (en plus de l'élément primitif α utilisé) à l'ensemble des serveurs de données. Chaque serveur parcours alors tous les enregistrements de sa case. Pou chaque enregistrement P

(p''_1,..,p'_n), la recherche s'effectue sur la zone non-clé. Algorithme 7- 1 montre le traitement effectué par chaque serveur de données pour chaque enregistrement lors d'une recherche d'une chaîne séquentielle en utilisant les signatures cumulatives.

1- Calcul de S_l au préfixe P' de longueur l.
Si S=p''_l alors l'enregistrement est sélectionné. Fin de la recherche.
2- 2.0- $S'=S$; $S_c=S_l$; i=1;
2.1- $S_c := p'_i$ XOR p'_{l+i}
2.2- S' :=*Antilog* (log S'+1)
3- Si $S' \neq S_c$ alors i :=i+1 ; revenir à (2) puis à (3) jusqu'au dernier p'_i de P'.
4- Si $S' \neq S_c$ alors passer à l'enregistrement suivant.
Sinon l'enregistrement est selectionné

Algorithme 7- 1 : Recherche d'une Chaîne Partielle en utilisant les Signatures Cumulatives.

L'algorithme commence par la lecture de p''_l. Ensuite, cette valeur est comparée à la signature reçue S. Dans le cas d'égalité, l'enregistrement est sélectionné. Le calcul de signatures algébriques s'applique successivement aux l caractères suivants jusqu'à ce que le serveur de données trouve la chaîne recherchée ou on examine sans succès le dernier caractère de la zone. Une même procédure décrite ci-dessous est appliquée à chaque incrémentation $(i+1)$ dans l'enregistrement avec $S'=S$ initialement afin d'éviter le re-calcul de S'. L'incrémentation fait suite à échec de comparaison de signatures. . L'étape 2.1 de l'algorithme ci dessus soustrait alors le 1er caractère de la chaîne testée à la signature actuelle. Nous notons celle ci par S_c. Ainsi, pour l'élément pour tous élement à la position $l+i$, il suffit juste d'une substitution directe entre la signature du préfixe p_i et celle de p_{i+l} constituant S. L'étape 2.2 calcule $S'\alpha$.

Notons que cet algorithme ne fait à aucun moment le décodage de la chaîne sauvegardée au niveau de l'enregistrement. Cela est en respect avec notre but de protection contre les vues accidentelles des données.

Dans l'algorithme ci dessus, on a considéré que la partie non clé d'un enregistrement est inférieure à 255 octets. Si la recherche concerne une chaîne de plus de 2^f caractères, une opération *Modulo* est rajouté à l'étape 2.2 $S' :=Antilog\ (\log S'+1 \bmod 2^f-1)$.

La compléxité de recherche séquentielle utilisant les signatures cumulatives est de $O\ (n-l)$.

Exemple 7.4

Considérons la recherche d'une chaîne « *NICHE* » dans plusieurs enregistrements. Le client calcule la signature correspondante *S* et envoie le couple (*S*, 5). Chaque serveur de données examine les enregistrements constituant la case appartenant à ce serveur. Pour chaque enregistrement visité, la recherche commence par le calcul de la signature du préfixe S_5.

Pour l'enregistrement encodé *P* dont le contenu original non encodé est P « *CHIEN DANS SA NICHE DE CHINE* » (Figure 7- 3). Le calcul commence par la $Sign_\alpha$ (*CHIEN*).

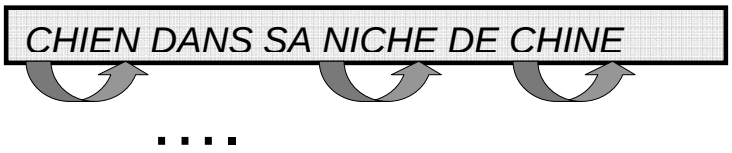


Figure 7- 3: Exemple de Recherche de Chaîne Partielle.

L'utilisation des signatures pré calculées cumulatives, permet de tester directement la signature de 'CHIEN' contenue dans p_5 ($Sign_\alpha$ (*CHIEN*)= p_7). Considérons que $S \neq S_5$, ce qui est le cas dans notre exemple puisque les signatures algébriques permettent de détecter une permutation au niveau des éléments. Un décalage à droite est fait. En utilisant les signatures cumulatives, le serveur exécute les opérations suivantes :

(i) $S_c = S_5$ $S'=S$;

(ii) $S_c = p''_1$ XOR $p''_6 = Sign_\alpha$ (C) XOR $Sign_\alpha$ (CHIEN).

(iii) $S' = Antilog\ (log\ S' + 1)$;

Apres l'étape (*ii*), le serveur produit la signature de '*HIEN* '. Pour chaque encodage, l'étape (*iii*) fait l'opération $S'=S\alpha$. A ce moment, il n'y a pas d'égalité ($Sc \neq S'$) et le serveur continue et boucle sur les trois étapes jusqu'à ce qu'il trouve '*NICHE*' sauf s'il y'a cas de collision. Si c'est le cas, le serveur détermine et résolue cette collision (on abordera la technique de détection dans les sections suivantes). Si le serveur continue la recherche au sein du même enregistrement, la signature de 'CHINE' ne sera pas équivalente à celle reçue par le serveur (propriété des signatures algébriques). Enfin, le serveur répond au client en joignant les coordonnées de l'enregistrement dans lequel '*NICHE*' a été trouvé.

7.2.3.2. Recherche de chaînes par la Méthode *n*-Grammes

Cette fonction se base également sur les propriétés de signatures cumulatives, [L&all06]. Un pré-traitement, comme pour l'algorithme de Boyer-Moore [BM77] par exemple, crée une table dite ici, *table de signatures de n-Grammes*. Un *n-Gramme* (ang. *n-Gram*) tel que monogramme, digramme, trigramme… ; n = 1,2,3… ; est une suite de symboles successifs (p_i…p_j tels que $j-i=n$) dans la chaîne recherchée et dans l'enregistrement. La table est utilisée ensuite pour générer des sauts lors du parcours de l'enregistrement visité. Ces sauts peuvent grandement accélerer la recherche.

7.2.3.2.1. Le principe

A la réception de la requête de recherche émise par l'application, le client encode la chaîne à rechercher P_k ($p_1, p_2,…p_k$) au niveau du client puis l'envoie aux différents serveurs de données. A ce niveau, un calcul de différentes signatures algébriques logarithmiques (*LAS*) est effectué sur chaque *n*-Gramme de la chaîne reçue. On appelle par *n-Gramme*, l'ensemble des sous chaîne $p_i…p_{i+n-1}$ avec $i=1,…, k-n+1$. La valeur de n ($n>=1$) est fixé à l'implémentation suivant les besoins de la recherche.
Le principe de cet algorithme consiste à comparer, à chaque fois, le dernier *n*-Gramme de la chaîne reçue à celui de l'enregistrement. Si ces deux *n*-Grammes ne sont pas égaux, des tests sont fait pour retrouver sa position (s'il existe) dans cet enregistrement. Un saut spécifique à cette position est fait à chaque fois. Si cette position n'est pas trouvée, le saut est le plus important. A la fin du traitement, si la chaîne est retrouvée, le serveur procède à une résolution de collision.
Dans ce qui suit, nous expliquons les différents étapes pour la recherche par la méthode *n*-Gramme.
La signature algébrique cumulative P'_k de chaque élément P_k est noté CAS (P_k). On note $p'_i=CAS$ (p_i).
Considérons $P_{k..1}=p_k...p_1$ avec $1<k<l$. Pour trouver la signature algébrique de $P_{l-k+1}=p_l…p_{l-k+1}$, il suffit de calculer: AS (P_{l-k+1})= (p'_l *XOR* p'_{k-l})/ α^{k-l}. Cette propriété nous permet d'avoir la signature algébrique sans avoir à visiter et parcourir l'enregistrement. Dans GF, cette signature algébrique s'écrit alors :

$$AS\ (P_{l-k+1}) = Antilog\ [Log\ (p'_l\ XOR\ p'_{k-l}) - k+1)\ \text{Mod}\ 2^f-1]$$

7.2.3.2.2. Pré Traitement

Soit P'_k la CAS (ang. cumulative algebraic sisgnature) de la chaîne P_k à rechercher. L'algorithme commence par le pré calcul de la signature algébrique logarithmique LAS (ang. Logarithmic Algebraic Siganture) du dernier *n*-Gramme de P_k (le plus à droite). Log AS (P_k)=LAS ($p_{k-n+1}..p_k$). Mettons la valeur obtenue dans une variable V.

- Pour tous les autres *n*-Grammes de P_k, on met les valeurs de leurs *LAS* après hachage dans une table *T* appelée *table des n-Grammes*. Celle ci est construite de la manière suivante : T à L entrées T [0,…, L-1]. Fixons L à k+2δ avec δ choisi arbitrairement comme δ= Int (0.25K). Si p est un *n*-Gramme, hachons *p* dans T (i) tels que $i=AS\ (p)\ Mod\ (k+\delta)$. Le choix de δ est de telle sorte qu'on a le moins de collision dans les entrées de notre table *n*-Grammes T [0,…k+δ-1]. On utilisera les dernières δ positions (entrées) pour les débordements dans la table (ayant la même valeur de hachage). Ces débordements nécessitent par la suite, un traitement de résolution de collision. Chaque entrée T (i) est représentée par un triplet noté (s, p, d) :

- *s* est la *LAS* du *n*-Gramme (on commence par la droite) dans P_k haché en T (i). Différentes signatures *LAS* pouvant avoir une même valeur de hachage i. par contre, une même valeur pour des *n*-Gramme différents correspond à une erreur de recherche.
- *p* est l'offset du *n*-Gramme haché avec le respect de l'ordre et du denier *n*-Gramme.
- *d* est mis à 0 à l'état initial. Il pointe vers le prochain débordement et induit une collision. T (i+d) contient alors le *LAS* du *n*-Gramme p' haché en T (i) (contenant déjà la *LAS* du n-Gramme *p*). T (i+d) contiendra alors une valeur différente de celle de la valeur *LAS* de *p*.

7.2.3.2.3. Recherche de la Chaine

On décrit le processus pour un enregistrement R de longueur M. Soit R_i^n le *n-Gramme* dans *R* finissant par p_i. On commence des *n*-Grammes par la comparaison des *LAS* à $i=k$. On teste alors *LAS* (R_k^n) à la variable *V*.

Dans le cas ou *LAS* (R_k^n)= *V*, on teste l'égalité cumulative. Si celle ci est vérifiée, on passe à la résolution de collision. La recherche finit avec succès s'il n'y pas de collision. En cas d'inégalité, on procède au hachage de *LAS* (R_k^n) par une fonction qui retourne *i* l'index d'entrée dans la *table n-Gramme*. A cette position, on cherche, dans la table *n*-Gramme, la signature logarithmique *LSA* qu'on vient de comparer. Si on ne trouve pas cette signature nulle part dans la table, un saut de $k-n+1$ est effectué dans *R*. On compare alors *S* au digramme suivant $p'_{k-n+1}\ldots\ p'_{2k-n+1}$. Dans le cas contraire (signature *LSA* trouvé dans la table), un saut de $p\ (j)$ position est fait dans l'enregistrement. La valeur j représente l'index d'entrée (position) ayant donné l'égalité dans la table. Les sauts sont plus petits d'une position vu que le dernier *n*-Gramme ne figure pas dans la table mais dans la variable *V*. Dans notre implémentation, ce saut est incrémenté de telle sorte qu'il est directement pris en compte lors d'un décalage.

Ces étapes sont répétées chaque *n*-Gramme traité dans l'enregistrement. A chaque fois, la signature logarithmique du dernier *n*-Gramme est comparée à *V* dans l'enregistrement. Le traitement s'arrête au succès de la recherche ou lorsque le dernier décalage atteint la fin de l'enregistrement (dernier suffixe). La vitesse de recherche dépend du nombre de comparaisons des *LAS*s et de la taille de l'enregistrement. Supposons *m* la longueur de celui ci. Le temps de complexité pour la recherche d'une chaîne de caractère de longueur *k* est en général *O (M-k /k-n+1)*. Pour notre implémentation, on utilise des *n*-Grammes tels que *n=2*. Ces *n*-Grammes sont appelés *digrammes*. Ils s'avèrent suffisants. La complexité de recherche est alors de *O (M-k/k-1)*.

Dans le chapitre 8, nous présentons quelques mesures de performance puis nous les analysons. Nous montrons que cet algorithme est notamment efficace pour la recherche de longues chaînes. Notre technique est (k-n+1) fois plus rapide que le précédent algorithme de recherche utilisant les signatures algébriques cumulatives.

Notons la possibilité de collision entre les différents n-Grammes à travers leurs signatures *LAS*s. Ce cas est possible si deux différents n-Gramme possèdent une même signature hachée.

Cette collision rallonge la recherche en réduisant le saut moyen. Rappelons que suivant les propriétés d'une signature algébrique :a) - Si deux n-Gramme différent par seulement 1 seul caractère, la probabilité de collision est presque 0. Ceci est plutôt valable pour les signatures à *n* caractères ce qui n'est pas le cas ici. b)- la probabilité de collision pour n>1 est de 2^{-f} équivalent à *1/256* pour les caractères Ascii, EBCDIC… et de *1/64K* pour les caractères *Unicode*. Pour cette raison, les n-Grammes sur *n* caractères seront meilleurs. La probabilité devient $1/(256)^n$. Pour n=2, Les digrams tels que '*xy*' et '*yx*' ne provoquerons pas de collision à coups sur.

Exemple 7.5

On considère arbitrairement un enregistrement R= '*Université de Technologie Paris Dauphine*'.

Soit S la chaîne à rechercher S='*Dauphine*'. Figure 7- 4

On choisit n=2 ce qui équivaut à la recherche par digramme. On commence par le pré calcul des signatures logarithmiques de tous les digrammes de '*Dauphine'* excepté le dernier. La signature *LAS* du de celui ci (représentant 'ne') est calculée puis mise dans une variable *V*.

La longueur de *S* étant *k*=8 ici. La dimension de *T* est alors Dim [*T*]=12 qui est suffisant suivant le hachage *Mod* 2^3.

Les signatures *LAS* de digrammes de 'Dauphin' qui sont, suivant l'ordre dans T : 'in','hi', 'ph', 'up', 'au', 'Da', sont mises dans la table des digrammes *T*.

Nous considérons ici qu'il n'y a pas de collision sur ces digrammes et que ceux ci ne génèrent pas de mêmes valeurs de *LAS*.

(a) Universite de Technologie Paris Dauphine
Dauphine Dauphine Dauphine
Dauphine Dauphine Dauphine

(b) Universite de Technologie Paris Dauphine
Dauphine Dauphine Dauphine Dauphine
Dauphine Dauphine Dauphine

Figure 7- 4 : Recherche par n-Grammes dans un enregistrement avec n=2 puis n=1

La recherche utilisant les digrammes nécessite ici 6 tests et 5 décalages (Figure 7- 4a). Nous soulignons, dans cet exemple, les digrammes examinés et nous montrons les décalages successifs l'un après l'autre sur les 2 lignes de la figure ci dessus.

Les premiers tests de comparaisons de *LAS* de 'si', ' t', 'lo', 'ar' ne sont pas égaux à la *LAS* ('ne'). Leurs signatures n'étant pas aussi disponibles dans T. Un saut de 8-1=7 positions est généré à chaque fois dans l'enregistrement R. Le 5 ème test visite le digram 'up', celui ci ne concorde pas avec 'ne' comme les digrammes précédant mais la signature *LAS* de 'up' est cette fois ci disponible dans la table *T* à la position quatre (4). Le dernier digramme étant pris en compte pour cette position, un décalage de 4 positions est effectué alors dans l'enregistrement *R*, ce qui emmène à un alignement avec le suffixe de l'enregistrement. Un test sur le digramme 'ne' est effectué avec succès. Nous testons ensuite l'égalité cumulative puis procédons à la résolution de collision. La recherche se termine avec succès.

L'exemple de la Figure 7- 4b illustre la même recherche en utilisant un monog*ramme* (*n*=1). Cette recherche nécessite 7 tests. Notons que l'algorithme de Boyer-Moore nécessite le même nombre de tests. La recherche utilisant *n*=3 (trigramme) nous coûte 1 test de plus.

Ces exemples montrent que l'algorithme par les *n-Grammes* n'est pas statique en ce qui concerne le choix de *n* (surtout si la chaîne dans laquelle on recherche n'est pas grande). La valeur n=1 peut être meilleure si les symboles constituant la chaîne recherchée et le contenu de l'enregistrement sont discriminatifs. Généralement, un *n* plus large est préférable lorsqu'il s'agit d'un grand enregistrement. Un caractère (et même un digramme) peut être répété plusieurs fois dans la zone dans laquelle s'effectue la recherche. Un grand n (ou plutôt un bon choix de n) aboutit à une moyenne de k-n+1 saut.

Remarquons aussi que le calcul de *LAS* pour un *n-Gramme* dans *R* ou *S* coûte la même chose avec la variation de *n*.

7.2.4 Recherche du Plus Grand Préfixe Commun

Pour ce type de recherche, le client envoie la chaîne avec le préfixe donné par l'application à chaque serveur de données. Ces serveurs comparent la chaîne reçue avec le préfixe de chaque enregistrement contenant un même préfixe. Sur chaque serveur du fichier, la recherche parcours tous les enregistrements de la case de données. On sélectionne tous les enregistrements ayant le plus long préfixe commun avec la chaîne reçue, [LMS05a]. Chaque serveur envoie le résultat au client. Le client termine la recherche, en comparant les réceptions de chaque serveur, comme nous montrons plus loin. Pour simplifier la présentation de l'algorithme, nous là limitons au cas d'une page de données par enregistrement.

En premier, le client encode la chaîne *S* en $S''=s''_1s''_2\ldots s''_l$ et l'envoie à tous les serveurs. Algorithme 7-2 montre les différents traitements au niveau d'un serveur de données pour chaque enregistrement.

- **1. Trouver le 1er enreg tels que $p''_1=s''_1$. u= 1;**
- **2. Parcourir l'enreg P suivant i=2^u //Test si $p''_2= s''_2$…**
- **3. tant que $p''_i=s''_i$ et $2^u<n$ faire u:=u+1; L=u;**
  - **Si $2^u>n$ aller à (4)**
  - **Sinon Soit $j= (2^u + 2^{u-1})/2$**
    - **Si $p''_j=s''_j$ alors Recherche dans $[j,p''_u]$ L:=j**
    - **Sinon Recherche dans $[p''_{u-1},j]$**
    - **Revenir à (3)**
- **4. Rajouter P à la liste. Soit s''_L sa signature cumulative**
- **5. Examiner le prochain enregistrement:**
  - **Si $p''_L=s''_L$ alors u=L revenir à (2). (en multipliant par α^L)**
  - **Sinon passer à l'enregistrement suivant.**
    - **Déterminer nouveau L**

→ **Envoi de L & P au client (taille du plus grand préfixe et le (les) différents enregistrement(s) concerné(s)).**

Algorithme 7- 2 : Recherche du Plus Grand Préfixe

Chaque serveur trouve d'abord le 1er enregistrement R_1 tel que $p''_1=s''_1$. Le serveur compare alors p''_i à s''_i et tant que $p''_i=s''_i$, il progresse vers les autres caractères (i=0,1,.., l). L'algorithme basique l'aurait conduit au test de chaque valeur successive de i. L'utilisation de signatures cumulatives permet à un

échantillonnage de celles-ci. En effet, le résultat positif pour un i donné signifie le succès pour tous les i précédents avec une grande probabilité en général, dans les GFs employés. On peut donc faire l'économie de ces comparaisons. Ces considérations nous ont conduit à appliquer dans SDDS-2005 l'heuristique par la progression (recherche) binaire dichotomique bien connue. On avance dans la comparaison tant qu'il y'a égalité par le pas $i = (1,2,4\ldots2^l)$. Ceci, jusqu'à ce que l'on rencontre une inégalité. Alors on revient à p''_i qui est au milieu du dernier intervalle examiné des valeurs de i. Dans le cas d'inégalité de nouveau on revient en arrière d'une manière similaire etc. Autrement, on avance à p''_i au milieu de l'intervalle restant etc.

Soit L la longueur du préfixe ainsi trouvée. Le serveur retient L et passe à l'enregistrement suivant. Il commence alors la visite par le caractère p''_L. Cela est possible grâce à l'encodage en signature cumulative. Si $p''_L \neq s''_L$, alors le serveur passe à l'enregistrement suivant. L'économie de comparaisons jusqu'à L est alors un avantage évident sur l'algorithme basique.

Autrement, si donc $p''_L = s''_L$, alors le serveur recommence la progression dichotomique. Une plus grande ou la même longueur L' du préfixe commun peut en résulter. Le serveur passe au prochain enregistrement avec L mis à jour par la valeur de L' etc.

Une fois le dernier enregistrement visité, le serveur entreprend la *vérification de la collision*. Il se peut en effet qu'il ne s'agit que de collisions pour certains enregistrements sélectionnés. On décrit cette phase dans la section 7.3.

Enfin, chaque serveur envoie sa liste au client. Actuellement, chaque message contient L suivi par les clés des enregistrements concernés. Le client fait le tri dans cette liste et choisi L_{max}. Il sélectionne alors les enregistrements correspondants et retourne le résultat à l'application.

Exemple 7.6

Considérons qu'un client lance la recherche du plus grand préfixe avec une chaîne http://www.Ceria.daphinc.fr qu'il envoie aux serveurs de données. Considérons que toute partie non clé dans le fichier commence par une URL « *http://www* » dans chaque enregistrement. La recherche séquentielle suivant chaque caractère devient plus ou moins importante au bout de quelques enregistrements avec des parcours inutiles suite à des comparaisons identiques sur les premiers caractères. Au lieu de rechercher séquentiellement dans les enregistrements ayant une similarité de préfixe, le serveur applique la méthode dichotomique décrite plus haut. Cela accélère la recherche par rapport au parcourt séquentiel. Dans ce serveur, L= 11 ce qui équivaut à 11 comparaisons inutile.

Il commence la comparaison à L qui constitue alors un offset dans le parcourt des enregistrements suivants. Ce gain est plus important quand *L>>1*.

Exemple 7.7

Lors de la recherche du plus grand préfixe dans une table relationnelle des hôtels à Paris (classe, ville, code postal,..) contenant des enregistrements de type « **…… *: Paris :*……75016 » ayant une longueur de 30 caractères juste pour le nom de la ville « *Saint Denis Sur Seine Les Bains* ». La méthode dichotomique est aussi appliquée lorsque le client envoie le préfixe commençant par le nom de la ville par exemple.

Exemple 7.8

Supposons que le client envoie la chaîne *S*=’UNIVERSITE PARIS *DAUPHINE’* à deux serveurs contenant les enregistrement *P* et *P’* respectivement. Considérons la Figure 7- 5. La taille *L* du plus grand préfixe commun trouvé est de 10 sur le serveur de l’enregistrement *P*. Les différentes comparaisons testées dans *P* enregistrements sont illustrées dans Figure 7- 5. Cela nécessite 7 comparaisons. La méthode séquentielle aurait nécessité 11 comparaisons en plus du derbier test négatif de ‘D’. Le nombre de comparaisons est alors de 12..

On a par ailleurs *L* = 15 sur le serveur de l’enregistrement *P’*. Dans ce cas, trouver L nécessite 8 comparaisons au lieu de 16 par la méthode séquentielle.

Les deux serveurs envoient leurs réponses au client. Celui-ci sélectionne l’enregistrement avec *L*=15.

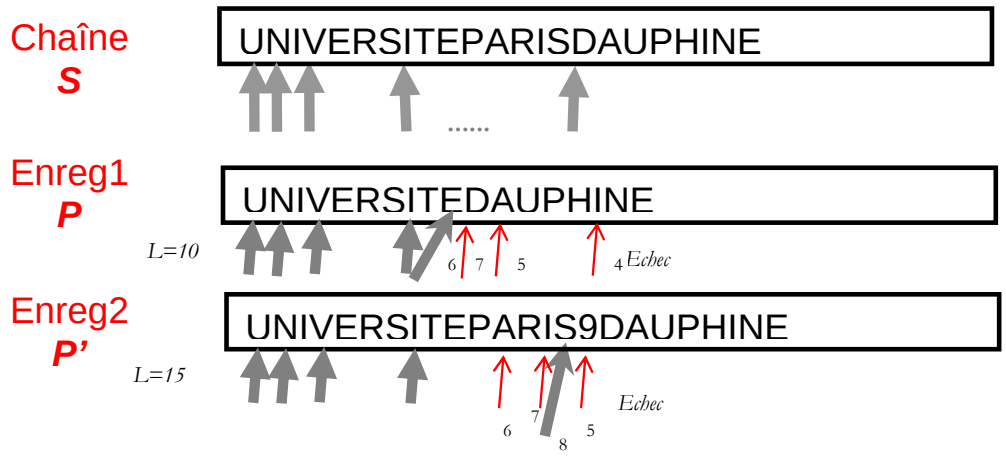


Figure 7- 5: Exemple de Recherche du Plus Grand Préfixe.

Enregistrement *P*: Comparaion (p et S).
Egalité sur p_1, p_2, p_4, p_8.
Inégalité sur p_{16}, p_{12}.
Egalité ur p_{10}.
Inégalité sur p_{11} → L=10

Enregistrement *P'*: Comparaion (p', S).
Egalité sur p_1, p_2, p_4, p_8.
Inégalité sur p_{16}.
Egalité ur p_{12}, p_{14}, p_{15}. → L=15

Figure 7- 6 : Comparaisons dans les enregistrements P et P' lors de la recherche du plus grand préfixe.

Si dans un autre cas, *P* et *P'* font partie d'une même case de données et si nous supposons que *P* est positionné avant *P'* (*P* de clé inférieure à *P'*), alors la comparaison dans *P'* commence directement à la position 10. Le serveur compare avec la position 11 puis 13 puis 17. Une inégalité est alors détectée, il retourne alors à la position 15 et la chaîne contenue dans *P'* est sélectionnée après le test sur la position 16. En précédant de cette manière, moins de comparaisons ont été utilisées par rapport à la méthode séquentielle commençant au début de l'enregistrement. Cela accélère la recherche.

Analyse

Les signatures cumulatives donnent à notre algorithme, par rapport à celui usuel, la possibilité d'une économie substantielle de comparaisons. D'abord, il s'agit du volet correspondant à chaque nouvelle visite, où on commence la comparaison par le caractère p''_L au lieu de p_1. On peut économiser jusqu'à L comparaisons par enregistrement. Ensuite il y a une économie due à la progression dichotomique au lieu de celle séquentielle. Ici on utilise $O((log_2 (L'-L))$ comparaisons à la place de $L'-L$. Pour des chaînes plus longues, telles que des URLs ou chaînes DNA, les avantages pourraient être essentiels.
Enfin, ajoutons que notre recherche a pour complexité O (1) dans le meilleur cas. Cette situation correspond à la situation ou aucun préfixe n'est trouvé dans la case. Le cas le pire serait celui où tous les préfixes trouvés ne sont que les collisions. Il ne semble pas utile de s'en préoccuper davantage. L'analyse plus fine, au delà de nos expérimentations au Chapitre 8, notamment de divers cas pratiques reste un objectif futur.

7.2.5 Recherche de la Plus Grande Chaîne Commune

L'application qui appelle cette fonction, opérant sur le fichier préalablement ouvert, passe au client la chaîne de caractère *S* qui est celle maximale recherchée. Le nom du fichier est également concerné par cet envoi. Le serveur retourne tous les enregistrements (identifiés par leur clés) dont la zone non-clé

contient la plus longue chaîne commune avec *S*, ainsi que cette chaîne. A noter que plusieurs chaînes communes différentes peuvent avoir une même longueur maximale.

Le client commence par encoder *S*. Pour simplifier la notation, on note désormais *S* la chaîne encodée. Le client envoie *S* ensuite à tous les serveurs du fichier. Chaque serveur recherche alors les enregistrements correspondants dans sa case. Il visite à cette fin séquentiellement tous les enregistrements de la case. L'algorithme de ce parcours que nous exposons ci dessous, [LMS05c], généralise celui précédent. Figure 7- 7 illustre son sa conception.

Soit $P = p_1 \ldots p_n$ le 1er enregistrement visité. Le serveur commence la visite par s_1 qui est le 1er symbole dans *S*. Il le compare à p_1. S'il y'a l'égalité, il commence la progression dichotomique, en préservant la valeur courante de *L*. Si la fin de *P* ou de *S* n'est pas atteinte pour *L* trouvé, le serveur continue l'exploration de *S*, à partir de s_{L+1}. La visite suivante est alors celle de s_{2L+1}. De ce parcours, le serveur sauvegarde la ou les chaînes communes les plus longues trouvées et *L* final.

Si $s_1 \neq p_1$ alors le serveur passe *à* s_2. Si $s_2 = p_2$, il effectue la comparaison similaire à celle décrite ci-dessus. Néanmoins, les chaînes communes trouvées d'une longueur inférieure à *L* ne sont pas considérées. Le serveur continue jusqu'à la fin de *S*. Puis, il recommence le calcul similaire pour p_2 et s_1 de nouveau. Etc jusqu'à p_{n-L}.

Le serveur passe alors à l'enregistrement suivant. Il procède de la manière similaire que pour le premier enregistrement, sauf que la comparaison initiale est celle de $s_L = p_L$. Il continue avec l'enregistrement suivant etc. Lorsque tous les enregistrements sont visités, le serveur vérifie les collisions (Section 7.3). Puis, il envoie la réponse au client. Chaque message envoyé contient *L,* la ou les plus grande(s) chaînes communes, et les listes des enregistrements sélectionnés. Le client finit la sélection, en ne retenant que les résultats avec L_{Max}.

Exemple 7.9

On considère la chaîne S = *'BIENVENUES LABORATOIRE CERIA DAUPHINE*' envoyée par le client à l'ensemble des serveurs de données. On considère deux enregistrements *P* et *P',* appartenant à la même case et sur lesquels on applique notre algorithme. Leurs contenus non encodés sont en Figure 7- 7.

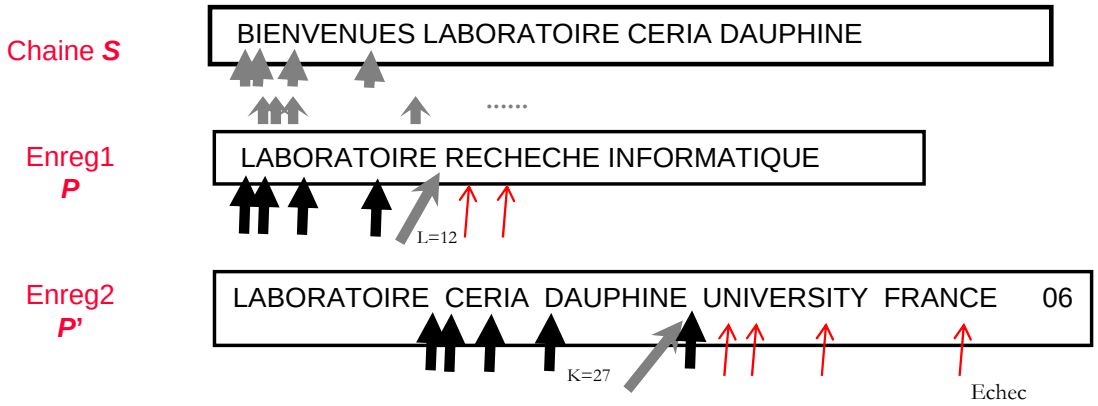


Figure 7- 7: Exemple de Recherche de la Plus Grande Chaîne Commune

Dans les deux cas, on commence par le premier caractère '*B*' de la chaîne *S* et suite aux inégalités avec le 1[er] caractère de l'enregistrement *P*, on avance dans *S* jusqu'à atteindre la caractère 'L'. La comparaison de celui ci avec le premier caractère de *P* est satisfaisante. Dans le premier enregistrement, suivant les égalités entre les caractères entre S et P, les comparaisons aux positions 1, 2, 4, 8 sont satisfaisantes. La comparaison à la 16 [ème] position échoue, le serveur revient alors à la position 12. A cette position, il y'a égalité. Cela n'est pas le cas à la position 14. Le traitement fini par retenir la taille *L*= 12 comme la taille de la plus grande chaîne commune actuelle. Le nombre de comparaisons ici est de 7 à partir de la comparaison du caractère 'L'. Notons l'existence de chaînes communes intermédiaires notamment lors de la comparaison du caractère 'E' de la chaîne *S*.

Dans le deuxième enregistrement *P'*, on commence directement par tester au niveau de la *12* [ème] position. L'égalité est vérifiée. On avance de 2, 4 puis 8 caractères Au niveau du 32 [ème] caractère, les caractères comparés ne sont pas égaux. On revient aux positions 24 puis 20. La position 16 est retenue (le ' ' après le *E* de *DAUPHINE*). La taille de la plus grande chaîne commune devient 27. Celle ci est '*LABORATOIRE CERIA DAUPHINE*' Cet enregistrement est retenu dans la liste envoyée au client avec une taille de *L*=27 Octets. Le nombre de comparaison ici est 10.

Analyse

La complexité de calcul d'une chaîne commune de longueur *L* est comme auparavant $O(log_2(L))$. Dans le meilleur cas, la complexité de recherche est $O(1) + 1/N$ où N représente le nombre d'enregistrements dans la case. Ce cas se présente si la totalité de la chaîne reçue constitue un préfixe du premier enregistrement sans qu'aucun autre enregistrement n'ait ce préfixe. Dans le pire cas, cette complexité de recherche est $O(n*l)$. Ici, *n* représente la taille de la zone non clé et *l* la taille de la chaîne.

envoyée par le client. Ce cas correspond à la situation ou aucun enregistrement ne partage même pas un seul caractère avec la chaîne reçue. C'est également la complexité de la méthode naïve. La complexité de notre méthode pour un fichier réel semble difficile à trouver analytiquement. Elle reste un objectif futur. Dans le chapitre 8, nous donnons quelques mesures pour la recherche de la plus grande chaîne commune.

7.3 Résolution de la Collision dans SDDS-2005

Dans SDDS-2005, le test de collisions est fait sur le serveur si la chaîne lui est connue et sur client autrement. Ce dernier cas concerne la recherche par préfixe et celle séquentielle d'une chaîne, [LMS05c].

7.3.1.1. Résolution sur le client

Cette approche est appliquée à la recherche par préfixe et à celle séquentielle d'une chaîne. Les réponses de serveurs peuvent contenir les collisions. Le client exécute sur les réponses reçues l'algorithme basique, 'caractère par caractère'. S'il découvre une collision dans une réponse unique, il renvoie la requête à tous les serveurs. Cette fois ci, le serveur effectue la recherche par l'algorithme basique.

7.3.1.2. Résolution sur le serveur

Dans cette approche, chaque serveur de données procède au contrôle de la collision après la sélection des enregistrements. Le contrôle se fait en parallèle sur les serveurs. Chaque serveur compare séquentiellement l'enregistrement (ou une partie de l'enregistrement) sélectionné à la chaîne de caractères reçue initialement. La découverte d'une collision relance la recherche par l'algorithme basique correspondant. Cette approche est donc lourde, mais les collisions devraient être rares en pratique. Dans SDDS-2005, cette approche s'applique à la recherche du plus grand préfixe ou de la plus grande chaîne commune. Ainsi, sauf dans le cas d'une collision unique, on évite le renvoi de la requête du client vers l'ensemble des serveurs de données. Aussi, dans le cas de plusieurs réponses, la vérification est faite en parallèle sur l'ensemble des serveurs.

7.3.2 Autres Approches à la Résolution de Collision

La résolution de collisions pourrait être entièrement sur le client. L'avantage serait le déchargement de cette tache pour les serveurs de données. L'inconvénient par contre, réside dans le fait que le client

traite plusieurs réponses par la méthode basique dans le cas ou le client reçoit des réponses provenant de plusieurs serveurs. Pour notre part, le choix de la résolution dépend du type de recherche de chaînes De techniques de résolution de collision peut-être plus rapides que la comparaison basique dans notre contexte peuvent être envisagées. Elles restent parmi les objectifs futurs. Citons d'abord celle où (1) on calcule la signature à 2 ou 4 symboles pour la chaîne recherchée (où sa plus grande partie commune trouvée). Puis, on calcule ces signatures pour chaque résultat final sur le client et on compare les signatures. La probabilité de collision sur les signatures plus larges devient négligeable.
Une autre méthode prometteuse peut-être dans le cadre de signatures cumulatives pourrait être appelée *par échantillonnage de n-grammes*. On compare alors pour chaque résultat sur le client, un échantillonnage de *n*-grammes, sélectionnés donc chacun efficacement. La probabilité de collision devient arbitrairement petite, pour *n* et le nombre d'échantillons grandissant. Cette approche semble mieux applicable aux chaînes longues.

7.4 Rappel sur d'Autres Algorithmes de Recherche de Chaînes

Il y a de très nombreux algorithmes de recherche de chaînes, [CL00]. Nous rappelons brièvement les principaux. Nous détaillons celui de Karp-Rabin [KR77] car il a une certaine similitude de conception avec notre algorithme séquentiel. L'autre raison est que nous comparons nos résultats expérimentaux dans le Chapitre 8. Nous avons conçu aussi une méthode alternative de recherche de chaînes par les signatures algébriques, également un peu similaire à l'approche de Karp-Rabin, que nous détaillons aussi. La comparaison expérimentale a montré néanmoins la plus grande vitesse de la recherche par les signatures cumulatives. Ainsi, cette méthode dite 'par la méthode 'XOR' n'a pas été incluse dans notre prototype final.

En ce qui concerne les algorithmes largement connus, il y d'abord celui *basique*, déjà mentionné (Figure 7-8). On y compare les chaînes symbole par symbole. Il a souvent une complexité au pire de *O (M*N)*, ou *M* représente la taille de la chaîne recherchée et *N* la taille de la chaîne dans laquelle se fait la recherche [LC00]. Cet algorithme a fait l'objet d'améliorations progressives. En 1970, Cook a démontré l'existence d'un algorithme en *O (M+N)* au pire cas. Puis, Knuth et Pratt ont proposée une solution encore plus pratique peu de temps après. Parallèlement, Morris améliora cet algorithme en 1976. L'algorithme final est connu sous le nom de *Knuth-Morris-Pratt*, [KMP77]. Entre temps, un autre algorithme fut découvert par Boyer et Moore, [BM77]. Cet algorithme est souvent considéré comme le plus rapide en pratique en général.

Les deux algorithmes nécessitent des temps de prétraitement qui peut être important pour des chaînes longues. En 1980, Karp et Rabin proposèrent un algorithme sans prétraitement [KR87]. Sa complexité est presque toujours *O (M+N)*.

Une anthologie impressionnante des algorithmes de recherche de chaînes est sur le site [CL00]. On peut y trouver la description et le code des algorithmes précités, mais aussi de Quick Search, de [CL03]. On y trouve aussi les simulations en Java. La figure montres le pseudo code pour la recherche d'une chaîne en utilisant l'algorithme naïf. Cette technique est très coûteuse. En effet, lors d'un échec dans une recherche, la position d'échec n'est pas explorée, ce qui oblige à refaire des comparaisons semblables qui auraient pu être évitées.

```
int BruteSearch (char *p, char *a) {
        int i, j, M = strlen(p), N = strlen(a);
        for (i = 0, j = 0, j < M && i < N; i++, j++)
        while (a[i] ≠ p[j]) {
        i -= j-1; j = 0;
        }
        if (j == M) then return i-M; else return i;
        }
```

Figure 7- 8 : Pseudo Code de Recherche par l'algorithme basique

7.4.1 Algorithme de Karp-Rabin

Le principe de l'algorithme de *Karp-Rabin* [KR87] est d'utiliser une fonction de hachage dispersée. Cette fonction, appliquée à une chaîne de caractères, produit une valeur de hachage semblable à la signature de cette chaîne à rechercher. En effet, le hachage permet d'éviter un nombre quadratique de comparaisons. Au lieu de tester la similarité à chaque position, on le fait qu'en cas de bon alignement entre les caractères des deux chaînes comparées. Puis, si les valeurs de hachage sont similaires, les chaînes sont semblables.

Le principe de l'utilisation d'une fonction de hachage comme signature du motif à rechercher réside dans la possibilité de ne retenir que quelques candidats pour les positions d'occurrence. On fait donc glisser une fenêtre de longueur *m* le long du texte dans lequel on recherche notre chaîne On détecte des positions candidates à occurrence en calculant une signature à chaque position de la fenêtre. Chaque caractère est alors représenté par sa valeur calculée suivant la fonction de hachage. Plus précisément, il s'agit d'associer un certain entier à chaque portion du texte de façon qu'il soit très improbable que deux portions différentes correspondent au même entier. Cet entier équivaut à une sorte de « signature » calculée de façon incrémentale.

Soit une sous chaîne $P(p_i, p_{i+1},.., p_{i+M-1})$. La chaîne P peut être considérée comme un nombre entier en base d (où d est la taille de la chaîne).
Pour chaque P_i, la fonction de hachage produit la valeur H_i (signature de P_i).s'écrit alors :

$$H_i = p_i\, d^{M-1} + p_{i+1}\, d^{M-2} + \ldots + p_{i+M-2}\, d + p_{i+M-1}$$

Les multiplications par 2 (d=2) sont fait par des décalages. L'utilisation d'un nombre premier assez grand évite les débordements suite à ces décalages. Cela dépend du hardware (modulo implicite).

A priori, calculer la fonction de hachage (ici, appelée aussi fonction d'adressage) semble aussi coûteux que d'effectuer toute la comparaison.
Pour remédier à cela, Rabin et Karp ont proposé une méthode qui permet de calculer la fonction de hachage de la sous chaîne commençant à la position *(i)* à partir de celle de la position *(i-1)*.

A la suite d'un échec de la comparaison à la position $_i$, la recherche continue en passant aux prochaines positions. En décalant la recherche d'une position vers la droite dans le texte (zone dans laquelle on recherche), il n'est pas nécessaire de parcourir (traiter) la totalité des caractères pour le calcul de nouvelle signature. Ainsi, pour la sous chaîne commençant à la position suivante, H_{i+1} est calculée à l'aide de l'expression $H_{i+1} = (H_i - p_i d^{M-1})*d + p_{i+M}$

En cas d'égalité entre les valeurs de hachage (signifiant la similarité entre deux chaînes de caractères), il est nécessaire de les comparer caractère par caractère pour des raisons de collision. Cela est du à l'utilisation de cette fonction de hachage (collision de valeurs).
En assimilant la fonction de hachage à notre formule de calcul de signatures algébriques. Le calcul de la signature d'un élément P (p_0, p_1,…, p_N) sera équivalente au calcul suivant :

$$Sign_\alpha(p_0, p_1,.., p_{N-1}, p_N) = \sum p_i\, \alpha^{N-i} = (\sum p_i\, \alpha^{N-i-1})\,\alpha + p_N \qquad i=0,..,n$$
$$= Sign_\alpha(p_0, p_1,.., p_{N-1})\,\alpha + p_N$$

Les deux dernières égalités montrent la propriété de calcul de signature d'un élément en fonction de la signature de l'élément précédent.

7.4.1.1. Calcul Incrémental des Signatures

Soit P une chaîne de caractères dont les élément sont $(p_0, p_1,...p_{m-1})$ de longueur m. Le calcul par la fonction de hachage H de la signature de P est donné par la fonction polynomiale suivante :

$$H(p_0...p_{m-1}) = (p_0\, d^{m-1} + ... + p_{m-1}\, d^0) \bmod q$$

Ici, d= 2^5 une base et *q* est un nombre premier plus petit que 2^{31}tels que $q= 2^{31}-1 = 2147483647$. On verra La raison d'utilisation de ce nombre dans les détails d'implémentation de cet algorithme. Pour une recherche partielle d'une chaîne, cette signature sera calculée suivant la manière suivante :

$$H(p_{i+1}..p_{i+m}) = ((H(p_i..p_{i+m-1})-p_i d^{m-1})d+p_{i+m}) \bmod q.$$

Coté implémentation, la formule incrémentale permettant ce calcul est donnée sous la forme suivante :

```
static long Sign_Polynomiale (char[] A, int m, int i, long h)
{
….
 }
```

Cette fonction possède différents paramètres d'entrées : la clé de hachage *h* correspondant à la portion du texte *A* de longueur *m* et commençant en position *i*. Elle renvoie la clé correspondant à la portion du texte commençant en position *i+1*.

Pour que cette procédure ait un faible coût, bien que cette tâche n'étant pas la partie la plus coûteuse de l'algorithme, on s'assurera que la puissance d^{m-1} n'est calculée qu'une seule fois. De même pour *modulo q*. Ces deux valeurs sont sauvegardées dans une variable de type *static*. On demande ici l'implantation la plus naïve possible, par *m-1* multiplications successives modulo *q*.

A la fin du traitement et en cas d'égalité, on vérifiera la similarité des deux chaînes comparées par la méthode naïve. Cela est du à la possibilité qu'il y ai collision. Notons que dans le cas le plus défavorable, l'algorithme KR possède une complexité de recherche de *O (m+n)*, *m* étant la taille de la chaîne à rechercher et n la taille de l'enregistrement dans lequel on recherche.

7.4.1.2. Implémentation de l'algorithme de Karp-Rabin

Différentes versions d'implémentations de l'algorithme Karp-Rabin existent. On cite notamment la version implémentée par Crochemore & Lecroq [C97, CL03, CL03a]. Le pseudo code de la Figure 7-9 présente l'implémentation d'une recherche d'une sous chaîne $A(a_1,..a_m)$ dans une chaîne de caractères $P(p_1,..p_n)$ de longueur *m* et *n* respectivement.

On calcule les valeurs de hachage h_a et h_p pour les chaînes *A* et *P* respectivement. Une base d=2 est utilisée et la multiplication par 2 est faite par des décalages à gauche. Lors du calcul de d^{m-1} et lorsque *m* dépasse 32, cela génère des pertes de ces caractères (multiplication par 0). Seuls les *d* (32 au max) caractères à droite seront pris en compte. Cela est du à la taille du registre sur les Pentium courants.

```
Define REHACH (a, b, h)  ( ((h – (a)*d) <<1) + b)


Void KarpRabin (char *a, int m, char *p, int n)
{  int d, ha, hp, i, j;
   /* Pré calcul */
    For (d= i= 1; i<m; ++i)
    d=(d <<1);                    // calcul de d=2^(m-1) avec des décalages à gauche

     for (hp=ha=i=0 ; i<m ; i++){
               ha= ((ha<<1) + ai);
               hp= ((hp<<1) + pi) ;
               }

   /* Recherche */
    j=0 ;
   while (j <= n-m) {
               if (ha==hp && memcmp (a, pj, m)   Output (j) ;
               ++j;
               hp= REHASH (pj, pj+m, hp);
               }
}
```

Figure 7- 9 : Pseudo-Code implémentant l'Algorithme de Karp-Rabin (version Crochemore)

Pour pouvoir comparer nos résultats, nous avons implémenté l'algorithme Karp-Rabin. Nous nous sommes basé sur la version de code (Figure 7- 10) développé par l'équipe du Professeur Ingold [I04] à l'université de Fribourg en Suisse. Cette version d'implémentation de l'algorithme Karp-Rabin montre la recherche d'une sous chaîne de caractères A de taille m dans une chaîne *P* de taille n. On ne retrouve plus la limite discutée ci dessus. L'écrasement de caractères dans la méthode utilisée par *Ingold* est résolu par l'introduction de l'opération '*Modulo*'. Cet algorithme ne limite pas la taille de la chaîne recherchée à 32 Octets. La multiplication se fait alors par 2^5 à chaque fois (d=32). Dans cette version de code, on utilise une grande valeur pour *q* (tels que $q=2^{31}-1$). Lorsque la taille des caractères dépasse 32, le '*Modulo*' est systématiquement appliqué et la totalité des caractères interviennent effectivement dans le calcul de la valeur de hachage. L'inconvénient ici reste le long calcul lorsque le '*Modulo*' est effectivement utilisé. On devrait alors s'attendre à des temps plus importants lorsque la taille de la chaîne recherchée dépasse 32 Octets quand nos signatures permettent la recherche d'une chaîne jusqu'à 255 Octets sans emploi du 'Modulo'. Autrement, seules deux opérations 'Modulo' sont effectuées

Ici, si les signatures h_a et h_p des chaînes comparées sont égaux, une comparaison caractère par caractère est alors nécessaire afin de résoudre les collisions probables.

```
#define q 33554393=2^25
#define d 32
int KarpRabinSearch (char *p, char *a)
{
    int i,  h_p = 0, h_a = 0; int n = strlen(P); m = strlen(A);
    D = d^(M-1) %q

    for (i = 0; i < N; i++) {
                              h_p = (h_p*d+p_i) % q;
                              h_a = (h_a*d+ a_i) % q;
                             }

    for (i = 0; (h_p ≠ h_a) && memcmp(p_i, a_i, n)==0; i++) {
                              h_a = ((h_a+d * q) - a_i *D) % q;
                              h_a = (h_a*d + a_(i+M)) % q;
                              if (i > n-m) return m;
                             }
    return i;
}
```

Figure 7- 10 : Pseudo Code implémentant l'Algorithme de Karp-Rabin (version Ingold)

Exemple 7.10

Prenons *A*= "*namental*" et *P*= "*na ental*", associons à chaque caractère son code ASCII. On note a[i] le code ascii du i ème élément de *A* et h_a la valeur qui résulte de l'application de la fonction de hachage sur cet élément. De même pour P[j], h_p est la valeur 'haché' de *P*. La comparaison de *A* et *P* produit les valeurs suivantes :

a[0] = 110, h_a = 110,	p[0] = 110, h_p = 110
a[1] = 97, h_a = 3617,	p[1] = 97, h_p = 3617
a[2] = 109, h_a = 115853,	p[2] = 32, h_p = 115968
a[3] = 101, h_a = 3707397,	p[3] = 101, h_p = 3711077
a[4] = 110, h_a = 17973635,	p[4] = 110, h_p = 18091395
a[5] = 116, h_a = 4731755,	p[5] = 116, h_p = 8500070
a[6] = 97, h_a = 17198685,	p[6] = 97, h_p = 3567213
a[7] = 108, h_a = 13487740,	p[7] = 108, h_p = 39487740

La valeur de hachage de *A* est différente de celle de *P*. Ces valeurs étaient égales avant le traitement le *3* ème caractère dans chaque chaîne Cet exemple prouve bien qu'un seul caractère induit des différences au niveau des signatures de chaînes comparées.

7.4.2 Recherche Partielle de Chaînes par la Méthode *XOR* dans GF (2^{16})

Cette fonction petmet égalemnt la recherche de chaines de caractères [LS04, ML06]. Le client calcule la signature algébrique de la chaîne à rechercher. Il calcule également la valeur du '*XOR*' appliqué à l'ensemble des caractères de cette chaîne. Supposons cette valeur *X*. La valeur de la signature

algébrique S, la valeur X ainsi que la taille L de cette chaîne sont envoyés aux serveurs de données. Rappelons que pour $GF(2^{16})$, la signature algébrique ici est composée de 2 symboles (S_1S_2 tel que S sur 4 Octets). La recherche ici concerne une chaine de caractère inférieure à 2^f-1 sinon l'utilisation d'un opérateur Modulo s'impose. L'Algorithme 7- 3 permet la recherche partielle par la méthode XOR et les signatures algébriques. Supposons X la valeur *XOR* de la chaîne à rechercher, S la signature de cette chaîne et L la taille de cette chaîne, N la taille de l'enregistrement P (p_1,…p_N) dans lequel on recherche cette chaîne. Ce traitement est fait au niveau du serveur pour chaque enregistrement.

i=*l* ;
1- Tant que i<N-*l*
Calcul de X'=*XOR* (p_i,….,p_i+*l*)
Si X=X' alors passer à (2).
Sinon i=i+1 ; retour à (1) ;
2- For (j=i ; j<=i+*l*;i++)
***S'* ^ =*Antilog* [i+ p_i];**
3- Si S' ≠ S alors i :=i+1 ; revenir à (1) jusqu'au dernier p_i de R.
Sinon R est selectionné (la chaine S est trouvée).

Algorithme 7- 3 : Recherche Partielle de Chaines par XOR et les Signatures Algébriques.

A la réception de ce message au niveau des serveurs de données, le serveur calcule d'abord la valeur '*XOR*' des premiers L caractères de cet enregistrement. Il compare cette valeur à la valeur X reçue. On commence par la comparaison de X vu que l'opération '*XOR*' est très rapide à calculer au sein d'un processeur. S'il y'a inégalité entre les deux valeurs, on conclut que les premiers L caractères de l'enregistrement ne satisfont pas cette recherche. Le serveur traite alors les L caractères suivants par un décalage d'une position à droite. Si par contre les valeurs '*XOR*' sont égales, on passe à la comparaison des valeurs de signatures algébriques. Le serveur calcule la signature algébrique de ces L caractères puis la compare à S. Si le premier mot (signature $Sign_\alpha^1$) n'est pas égal à S_1 reçue, il n'est pas nécessaire de continuer la comparaison, on passe alors à la position suivante. Dans le cas contraire, on teste l'égalité entre S_2 et signature $Sign_\alpha^2$. A l'égalité, l'enregistrement est rajouté à la liste de ceux contenant la chaîne recherchée. On passe au traitement du prochain enregistrement jusqu'à atteindre la fin de la case de données.

Exemple 7.11

Supposons que la chaîne recherchée est « *Ceria* ». L'enregistrement P_1 contient la chaîne ' *Universite paris Dauphine*'. L'enregistrement P_2 contenant la chaîne '*Laboratoire Ceria Dauphine*' (Figure 7- 11).

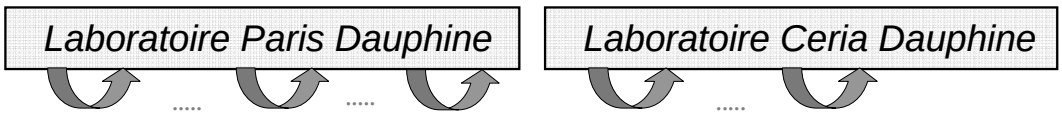


Figure 7- 11 : Recherche Partielle par les Signatures Algébriques dans R_1 et R_2

Le client commence par le calcul de *X* tel que *X*= XOR (*CERIA*). Il calcule également la valeur de la signature algébrique *S* de cette chaîne Dans un message aux serveurs de données, il envoie *X*, *S* et la taille *L*=5 de la chaîne à rechercher.

Supposons que la case de données contient deux enregistrements P_1 et P_2 de tailles 24 et 26 Octets respectivement. Lors du parcourt de P_1, le serveur calcule 19 valeurs '*XOR*' sans qu'il trouve d'égalité. Ainsi, trouvant une inégalité entre *X* et *X*'=*XOR*(Labor), il passe à la comparaison sur les L caractères suivants (abora). La valeur X' suivante est obtenue de la manière suivante : X'=X' *XOR* 'a' *XOR* 'L'. La comparaison continue jusqu'à atteindre le dernier caractère de P_1.

Lors du parcourt du deuxième enregistrement, il calcule 12 valeurs *XOR*. Au niveau de la 12 ème comparaison, il trouve égalité entre *X et X'*. Le serveur passe alors au calcul de la signature algébrique. Celle ci est obtenue en appliquant une opération *XOR* entre la signature de '*a*' et celle de '*C*'. Une division est faite par la suite par a^{12} (propriétés de signatures algébriques). Ceci permet d'obtenir la signature algébrique de '*Ceria*'. Le calcul concerne aussi bien $Sign_\alpha^1$ que $Sign_\alpha^2$. Si cette signature est égale à *S* (ce qui est le cas ici), l'enregistrement P_2 est sélectionné. Celui ci contient la chaîne recherchée. Le serveur de données envoie alors un message au client indiquant la clé de l'enregistrement P_2.

7.5 Synthèse

Les signatures cumulatives permettent la protection contre les vues accidentelles de données au niveau des serveurs. Cela est possible grâce à l'encodage de données par les clients impossible à décoder involentèrement sur un serveur. Ces signatures possèdent également des propriétés permettant la recherche de chaînes de caractères dans le contenu encodé. Nous avons présenté les différents fonctions de recherche de chaînes encodées dans la zone non clé que nous avons implémenté dans

SDDS-2005. Nous avons également montré que les algorithmes correspondants devraient être très efficaces. Nous montrons dans le chapitre 8 des résultats expérimentales confirmant ces conclusions. En particulier, nos algorithmes apparaissent en général ou au moins dans certains cas plus rapides que les algorithmes équivalents connus. C'est le cas notamment de la recherche par préfixe et celle du plus grand préfixe commun. Nous montrons dans le chapitre 8 que cela peut être aussi le cas de la recherche séquentielle de chaînes en utilisant les signatures cumulatives par rapport à celle également séquentielle par une version connue de l'algorithme de Karp-Rabin. Ensuite, ceci semble vrai pour la recherche par *n*-Grammes par rapport à l'algorithme de Boyer-Moore ainsi que pour la recherche de la plus grande chaîne commune par rapport à l'algorithme basique connu. Nous insistons toutefois que ces conclusions sont préliminaires. Ils demandent une vérification expérimentale et théorique approfondie. Celles-ci restent parmi les objectifs futurs indiqués dans le Chapitre 9. Notons que l'analyse théorique approfondie de nos fonctions semble d'une grande complexité.

Chapitre

8 *Mesures Expérimentales de Performances*

Dans ce chapitre, nous rapportons les résultats des différentes expérimentations sur notre prototype SDDS-2005. L'objectif de ces expérimentations est d'évaluer nos choix architecturaux et les performances pratiques de différentes fonctions implémentées. Notamment, les temps réels de réponse, en général impossible de calculer entièrement analytiqument. Dans une première section nous décrivons notre environnement expérimental. Puis, dans les sections suivantes, nous rapportant et discutons les performances des scénarios suivants :

- Calcul de signatures algébriques.
- Sauvegarde et restauration de fichiers SDDS.
- Mise à jours concurrentes des enregistrements.
- Calcul de signatures algébriques et cumulatives.
- Différents types de recherche par le contenu des enregistrements SDDS.
- Recherche de chaînes avec la méthode *n*-Gramme.

Pour valider nos résultats, des comparaisons sont faites à chaque fois, aux travaux existants notamment :

- Les signatures SHA-1.
- L'algorithme de Karp-Rabin.

On étudie tout particulièrement la scalabilité de nos opérations. Différentes expériences sont réalisées. Elles font l'objet d'analyses, suite auxquelles des conclusions seront tirées

8.1 Environnement Expérimental

Le prototype SDDS-2005 a été réalisé dans un environnement Windows 2000 et Windows *NT*. Il a été implémenté en langage *C* sous Microsoft Visual *C*++ 6.0. Différentes fonctionnalités offertes par l'*API WIN32* ont été utilisées.
Les expérimentations ont été réalisées sur un réseau local Ethernet 1Gbits/s reliant 6 machines Pentium IV 1.7*GHZ* de RAM et deux machines Pentium III 500*MHZ* et fonctionnant tous Windows 2000 Server.

Nous précisons la forte dépendance des résultats de ces expérimentations par rapport à la puissance des machines utilisées (vitesse du processeur, mémoire RAM disponible…) et la nature du réseau dont dépensent les temps de communication. Ces derniers dépendent de deux facteurs que sont *(i)* le *débit*, étant la quantité d'information transmise par unité de temps et *(ii*) la *latence*, étant le délai d'accès au site distant.

L'emplacement des serveurs et des clients peut aussi affecter les performances du système en raison des coûts de communication impliqués. Ce qui exige de faire un placement initial adéquat des serveurs, de sorte que ces derniers soient proches des clients potentiels. Les coûts de communication liés à la migration de données peuvent être excessifs et sa réalisation dans un environnement hétérogène peut être difficile. Les résultats sont enregistrés dans un fichier texte. Ce fichier garde, pour chaque paquet, les résultats concernant le nombre de requêtes émises et le temps total de traitement associé. Ce fichier permet de construire des tableaux récapitulatifs et de représenter des courbes de l'évolution des temps obtenus. Le prélèvement de ces résultats pour les analyser par la suite nécessite aussi des affichages au niveau des serveurs et des clients, pouvant ainsi ralentir le fonctionnement global du système. Le tableau 8-1 présente notre configuration expérimentale. Ces expérimentations sont faites en mettant en évidence le comportement de notre système face à une montée en charge d'où l'intérêt de l'implémentation de ces nouvelles fonctions.

Nous étudions le bénéfice de l'utilisation de nos techniques proposées basées essentiellement sur les signatures algébriques. Nous prélevons, à chaque fois, les temps nécessaires pour les différents traitements tout en variant différents paramètres intervenant dans le fonctionnement de notre système. Une comparaison est effectuée avec d'autres techniques, souvent connues dans le domaine correspondant Le Tableau 8- 1 qui suit résume d'autres détails de notre configuration :

Nombre de serveurs	Six serveurs numerates S0, S1, S2, S3, S4, S5.
Nombre de clients	Deux clients. L'une des machines serveurs est utilisée dans certaines expérimentations en tant que client et inversement.
Types de requêtes	Sauvegarde, restauration de fichier, Mise à jour d'enregistrements, recherches non clé.
Nature d'envoi des requêtes	+FC : avec contrôle de flux (ang. Flow control) - FC : sans contrôle de flux
Taille des messages	- 100 Octets pour les paramètres du message - de 100 Octets jusqu'à 64K pour les données suivant les expérimentations.
Paramètres de l'index	- Taille d'un nœud interne 80 Octets - Taille d'une feuille 100 Octets.
Mesures de temps	En millisecondes (*ms*) etparfois en secondes (*s*)
Mise à jour dans la case de données (MAJ du fichier).	+WC : Mise à jour avec changement (ang. with change). -WC : Mise à jour sans changement.
Taille des pages de données à l'intérieur d'une case.	16 KO
Tailles des symboles.	1 ou 2 Octets.

Tableau 8- 1 : Paramétres des des expérimentations

Les expérimentations faites par M. Ljungstrom [L00] ont montré que le calcul algébrique dans le corps de Galois : $GF(2^8)$ est plus performant que dans le corps de Galois : $GF(2^4)$. C'est ainsi que notre choix en corps de Galois s'est restreint à $GF(2^8)$ et $GF(2^{16})$.

Rappelons que le temps de réponse à une requête est réparti en trois composantes :

- Le temps d'envoi des paramètres de requêtes ainsi que leurs acheminements vers les serveurs de données (bande passante du réseau Ethernet).
- Le temps de traitement de la requête (configuration logicielle, vitesse processeur…).
- Le temps d'acheminement des réponses vers les clients.

La prise en compte de tous ces paramètres nous permet de mesurer l'impact du progrès technologique touchant la vitesse des processeurs et la bande passante du réseau sur les performances de notre prototype.

8.2 Sauvegarde d'un Fichiers SDDS en Utilisant les Signatures Algébriques

8.2.1 Temps de calcul de signatures pour un fichier de données

Dans cette section, nous nous intéressons particulièrement aux tests de sauvegarde de fichiers mappés de données sur disque. Le fichier mappé d'index est beaucoup plus petit en taille.

Nous procédons à la sauvegarde de différents fichiers pour étudier l'impact de la taille de ces derniers sur les performances. La requête de sauvegarde est envoyée par message multicast à l'ensemble des serveurs. En conséquence, un serveur de données qui reçoit cette requête ne s'occupe que de la sauvegarde et de l'envoi au client de l'intervalle de clés appartenant à ce fichier.

Nous étudions expérimentalement les performances de notre fonction de sauvegarde présentée au Chapitre 5. D'abord, Nous avons créé un fichier SDDS. Puis, nous avons varié le nombre d'enregistrements insérés. Nous avons étudié le meilleur temps physique de calcul de signatures à 2 symboles, sur 4 Octets au total. La taille d'un enregistrement avait été de 100 octets. Différents impléméntations du calcul ont été expérimentées. Par ailleurs, on fixe la taille des pages à 64Ko dans notre cas (GF (2^{16})) [M02]. Ces pages doivent être de tailles fixes pour des raisons d'uniformités de tests.

Le Tableau 8- 2 montre les résultats obtenus selon le nombre d'enregistrements insérés dans le fichier.

Nombre d'enregistrements	Temps de calcul des signatures algébriques (sur 2 Octets)
100	47
1000	453
2000	803
3000	1320
4000	1790
5000	2190

Tableau 8- 2 : Temps de calcul de signature sur *n* enregistrements.

La Figure 8- 1 montre la courbe correspondante. On remarque qu'elle est linéaire, comme on pourrait s'attendre. Ce qui peut surprendre par contre est que le temps par enregistrement n'est même pas constant, mais décroit légérement néanmoins. De 0.47 ms pour la petite page, à 0.438 ms pour la plus grande étudiée.

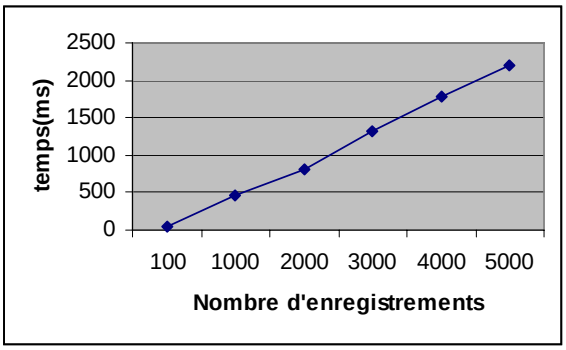


Figure 8- 1 : Calcul de Signatures Algébriques

Pour compléter ces mesures, nous avons mesuré le temps de calcul pour quelques tailles de pages spécifiques. Les temps sont de 31 *ms* pour une page de 1.6 MO ce qui approche les 20 *ms*/MO. Ce temps est 1.19 *ms* pour une page de 64 KO (19 ms/MO) et de 0.44 *ms* pour une page de 16KO, donc vers 27 *ms*/MO. Relativement en accord avec les résultats plus haut.

Notons aussi que des mêmes valeurs de données pour plusieurs enregistrements réduisent considérablement le temps de calcul des signatures (meme position dans la table *Antilog*). Cela facilite la tache au serveur qui n'a pas à recalculer, à chaque fois, la signature correspondante. Des valeurs aléatoires ont été donc utilisées pour le contenu de ces enregistrements.

8.2.2 Temps de Sauvegarde

Dans ces expériences, on sauvegarde sur disque une case dont on fait varier la taille. La 1ère requête du client provoque la sauvegarde totale (Chapitre 4). Puis, on redemande une sauvegarde, sans qu'une mise à jour ou une insertion soit faite. Dans d'autres requetes, le client effectue 5% de mises à jour dans les enregitrements à clé choisie aléatoirement. Puis, on fait une 2ème sauvegarde. Le tableau résume différents résultats mesurés. Les temps sont en *ms*. La tailel de page de données avai été de 64 KO, celle de la page de l'indexe avait été de 256 Octets.

Taille sur Disque (MO)	Nombre d'enregistre-ment insérés	Temps de calcul de signature	Temps de sauvegar de totale	Temps de la sauvegarde sans changements	Gain sauvegarde / temps global (%)	Temps de sauvegarde avec 5 % changement (ms)	Gain (%)
1.88	100	47	**562**	**50**	91.1	**65**	88.43
2.7	150	78	781	82	89.51	95	87.83
17.6	1000	453	5078	455	91.38	453	91.07
158	10000	4068	46406	4071	91.23	4085	91.19
393	25000	11003	117859	11003	91.33	11018	90.65

Lors d'une opération de sauvegarde initiale, la totalité du fichier est sauvegardée sur disque. On voit d'abord que le temps de calcul des signatures algébriques de pages n'est qu'un dixième de celui d'une sauvegarde totale (basique). Le temps légérement supérieur au temps de calcul des signatures seules durant la sauvegarde sans changement est dû à l'examen de la carte des signatures de la case. Nous rappelons que dans ce cas il n'y a en fait aucune écriture sur le disque. Le rapport entre les deux temps (gain relatif) est de l'ordre de 90 %. Il en est presque de même, un peu moins tout de même, pour la sauvegarde après 5 % de MAJ. Ici on a reécrit 5 % ou presque de pages sur le disque. On voit nettement ainsi l'intérêt de notre conception de cette fonction. Le gain devient naturellement moins important quand la proportion de changements dans la case augmente. Les insertions détériorent le gain plus rapidement pour un même taux, car elles provoquent les éclatements internes de pages de la case. Néanmoins, les expériences ont montré que l'insertion de seulement *5%* d'enregistrements, provoque le changement de 2 à 3 % de pages de la case [M02]. L'étude détaillée reste parmi les sujets prospectifs.

Ce gain reste le meme dans le cas d'insertion d'un nouvel enregistrement est rajoutée à la fin de la case de données sans provoquer d'éclatement de la case. le temps de la sauvegarde est équivalent au temps de calcul des signatures algébriques additionné au temps de sauvegarde d'une page de données (64 *KO* dans notre cas).

8.2.3 Scalabilité de Sauvegardes

Nous étudions la scalabilité de notre fonction de sauvegarde en répartissant un fichier SDDS sur davantage de serveurs. Le Tableau 8- 3 et la Figure 8- 2 montrent les mesures discutées plus haut en variant le nombre de serveurs de données sur lesquels s'étendent les enregistrements insérés.

Le gain en temps de sauvegarde est d'environs 6 à *7%* à chaque rajout d'un nouveau serveur [MLS03].

Le gain en temps de sauvegarde sur 3 serveurs est de 27% par rapport à un seul disque local.

Nombre d'enregistre-ments insérés	Temps de sauvegarde sur 1 serveur	Temps de sauvegarde sur **2** serveurs	Gain / 1 seul serveur (%)	Temps de la sauvegarde sur **3** serveurs	Gain / 1 seul serveur (%)
150	781	673	14	582	25.2%
1000	5078	4612	10	3700	27.3%
10000	46406	41328	11.2	35050	25%
25000	117859	110219	8.1	85362	27.1%

Tableau 8- 3 : Temps pour la sauvegarde d'un fichier SDDS sur plusieurs serveurs de données.

Pour la première ligne du tableau ci dessus, le calcul de la signature dans un seul serveur dure 78 *ms* tandis que pour 2 serveurs cela nécessite 48 *ms* (maximum des 2 temps de traitement sur l'ensemble des serveurs). En effet, supposons une case de données de taille 120. Une insertion de 150 enregistrements provoque l'éclatement de la case puis la migration des données vers la nouvelle case. On aura alors 60 enregistrements au 1 ^{er} serveur et 80 dans le deuxième serveur.

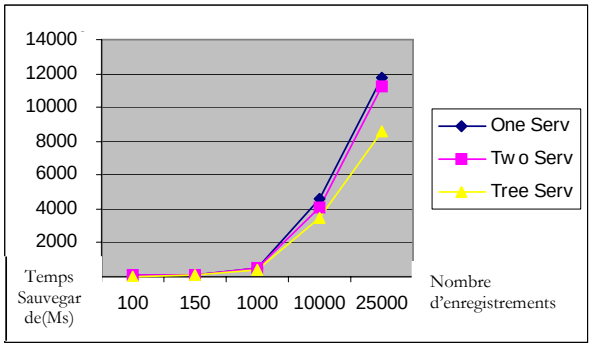


Figure 8- 2 : Scalabilité expérimentale de Sauvegardes.

La figure ci dessus montre l'évolution du temps de sauvegarde sur un, deux et trois serveurs. Si le fichier sera réparti sur un plus grand nombre de serveur, le temps de sauvegarde diminue. Ainsi, on pourrait s'attendre à des gains avoisinant les *50 %* quand le même fichier est réparti sur *5* serveurs. Cela nous permet d'affirmer que le module de sauvegarde offre des capacités de passage à l'échelle (scalabilité).

8.2.4 Sauvegarde de Fichier d'Index SDDS

Un fichier d'index est très réduit en taille par rapport au fichier de données (de 0.05 à 0.1%). En conséquence, le temps mis pour le calcul des signatures du fichier d'index est de l'ordre de $8.15\ e^{-5}$ (1%) par rapport au fichier de données (<<<1Ms). L'unité de sauvegarde sur disque doit être réduite à moins de *16Ko*. On fixe cette taille à *256* Octets (taille minimum pour la sauvegardesur un disque)

8.2.5 Variation de la Taille d'une Page

Des expérimentations ont été faites en variant la taille d'une page de données. Les résultats qui suivent montrent le temps de calcul de signatures dans une case de données. On insère ici 1500 enregistrements de 100 Octets chacun.

Tableau 8- 4: Temps de calcul des signatures algébriques /Variation de la taille des pages.

Taille de la page (Ko)	Temps de calcul de signatures algébriques (Ms)
8	657
32	651
64	640
100	659

Ce Tableau 8- 4 montre les temps de calcul de l'ensemble des signatures d'une case suivant des tailles différentes pour les pages de données. Ce temps décroît légèrement en fonction de la croissance de la taille des pages de données. Cela est logique puisque le processeur aura moins de signatures à calculer. Pour de pages de données de tailles supérieures (>64Ko), le processeur effectue à chaque fois l'opération du « *Modulo* » pour le positionnement dans la table *Antilog* ce qui explique les petites décroissances des temps de calcul. Pour des pages de taille 8 à 64 KO , le temps décroît de 3.5 %. Ce temps augmente de 3% lors d'utilisation de pages de 100 Ko par exemple. L'utilisation de petites pages permet d'avoir des temps plus réduits surtout lorsqu'il s'agit de petites mises à jour (insertion d'un seul enregistrement par exemple ce qui justifie l'utilisation de pages de 16Ko dans le prototype final de SDDS-2005. Le Tableau 8- 5 montre la sauvegarde moyenne pour une page de données.

Tableau 8- 5 : Temps de sauvegarde d'une page de données.

Taille de la page (Ko)	Temps de stockage d'une page (Ms)
8	14
16	15
32	15
64	16

8.2.6 Variation de la Taille d'une Signature Algébrique

Nous avons varié la taille de nos signatures algébriques. Nous rappelons que dans SDDS-2005, chaque symbole de la signature est sur 2 Octets, en tant qu'un élément de GF (2^{16}). Les tests dans le Tableau 8- 6 ont été réalisés en calculant l'ensemble des signatures d'une case de données de taille 17.6 *MO* ayant des pages de 16 *KO*.

Nombre d'enregistrements insérés	Longueur (Nombre d'Octets) des signatures	Temps de calcul des signatures (Ms)
1000	4	438
1000	6	453 (+4%)
1000	8	469 (+4.5%)

Tableau 8- 6: Temps de calcul des signatures algébriques / talles des signatures.

A chaque fois qu'une signature est calculée par rapport à une nouvelle puissance du polynôme primitif, le temps de calcul pour la signature globale est plus important. Cette croissance est d'environ 3 % à chaque rajout de caractère à la signature algébrique (1 ^{er} cas) et de 4.5 % dans le 2ème cas.

Nous rappelons que nous avons choisie la longueur de 4 Octets comme suffisante pour nos besoins étant donnée sa probabilité de collisions de 2^{-32} donc semblant suffisamment petite en pratique.

8.2.7 Sauvegarde d'un Fichier SDDS suite à un Eclatement

Supposons que le contenu d'un fichier soit sauvegardé sur disque. Puis, une insertion d'un enregistrement provoque un éclatement de ce fichier. Lors de la prochaine requête de sauvegarde, les pages de données seront complètement modifiées puisque la moitié des enregistrements sont déplacés vers un nouveau serveur. Le fichier sera mis sur disque comme s'il s'agit d'une première sauvegarde.

8.2.8 Utilisation de *GF (2^{16})* et Table *Antilog*

Lors du chapitre 4, nous avons montré que le dédoublement de la table *Antilog* permet d'éviter l'utilisation du '*Modulo*'. Ce calcul est potentiellement plus long, mais une table plus petite pourrait être entierement en cache L1. La table double avait été plus efficace. Cette option a donc été choisie pour SDDS-2005. Le gain experimental avait été d'environs 5 à 10% pour une case de 1Mo.

8.3 Comparaison avec d'Autres Travaux (Les signatures SHA-1)

Afin d'évaluer la performance de nos signatures algébriques, nous comparons nos résultats à ceux obtenus lors de l'implémentation de signatures SHA-1 [MLS03]. Nous donnons d'abord quelques détails d'implémentation de SHA-1 que nous avons réalisé à cette fin.

8.3.1 Implémentation de la Fonction de Hachage *SHA-1*

On suppose que l'on dispose d'une chaîne de caractère équivalente aux caractères d'une page de données. Rappelons qu'une fonction de hachage cryptographique [M02] est une fonction qui, pour tout message de longueur *m* arbitraire, calcule une « empreinte numérique » représentant une chaîne de longueur fixée (ici 160 bits). La Figure 8- 3 montre un exemple de calcul d'une signature SHA-1 :

- Message initial M= '01100001 01100010 0100011 01100100 01100101' de longueur *L*= 40.
- Rajout de 1 à la fin du message original. M' 01100001 01100010 0100011 01100100 01100101 1
- Rajout des 0 à la fin de M' en hexadécimal

M'= 61626364 65800000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000

- Rajout de 2 mots représentant la valeur *L* en hexadécimal 40= $(28)_{16}$. découpage en 16 blocs

M'= 61626364 65800000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000028

On définit les fonctions suivantes :

f_t (B,C,D)=(b AND c) *Or* (Not b) *AND* d 0<=t<=19 K=5A827999

f_t(B,C,D)=b *XOR* c *XOR* d 20<=t<=39 K=6ed9ebal

f_t (B,C,D)=(b AND c) *Or* (b AND d) *OR* (c AND d) 40<=t<=59 K=8f1bbcdc

f_t (B,C,D)=b *XOR* c *XOR* d 60<=t<=79 K=ca62c1d6

AND : et bit à bit. *OR* : ou bit à bit. *XOR* : ou exclusif bit à bit. *NOT* : complément à 1 bit à bit

Sn : rotation circulaire de n bits vers la gauche.

1-Initialiser *h0*=67452301, *h1*=efcdab89, *h2*=98badcfe, *h3*=10325476, *h4*=c3d2e1f0

2- Pour i= a jusqu'à n :

- Initialiser $w_t=m_{i,\,t+1}$ pour t=0,…,15
- Pour t=16 jusqu'à 79 $w_t=S^1(w_{t-3}\ XOR\ w_{t-8}\ XOR\ w_{t-14}\ XOR\ w_{t-16})$

Initialiser a=h0, b=h1,...e=h4

- Pour t=0 jusqu'à 79x=S^5(a)+ ft(b, c, d)+e+w_t+K_t, e=d, d=c, c=S^{30}(b), b=a, a=x, *h0*=h0+a,…, *h4*=h4+e

Le résultat sera alors *h(m)=h0||h1||h2||h3||h4*.

Figure 8- 3 : Exemple de Calcul de signature SHA-1

8.3.2 Mesures de Performances en utilisant les Signatures *SHA-1*

Le schéma standard *SHA-1*, implémenté sous *Visual Studio.net* produit une signature très sécurisée de *20* Octets. Nous utilisons ces signatures appelées, *Messages Digest* comme solution alternatives de signatures dans notre schéma. Nous les implémentons dans un but de comparaison avec nos signatures algébriques. Nous répéterons les mêmes expériences précédentes. D'après le tableau 8-8, nos signatures algébriques offrent un gain de 5% par rapport aux signatures cryptographiques *SHA-1*. Ce gain augmente avec l'insertion d'un grand nombre d'enregistrements dans la case de données. La méthode *SHA-1* est très utilisée dans les applications de cryptographie grâce à l'authentification parfaite des messages. Cependant l'utilisation de clés, inutiles dans notre cas, et surtout la consommation en ressources mémoire rend cette signature moins intéressante. En effet, nos signatures algébriques, avec seulement *4 Octets*, permettent une probabilité quasi nulle de trouver 2 pages différentes ayant la même signature.

Bucket size (Mb)	Number of record	Algebraic signature calculus (ms)	SHA-1 calculus (ms)	Storage time using SHA-1 (ms)	Storage time using alg. sign. (ms)	SHA-1 Store time for 5 % change (ms)	Alg. sign Store time for 5 % change (ms)
1.88	100	46	70	**602**	**562**	85	**65**
2.7	150	78	103	799	781	119	95
17.6	1000	438	680	5278	5078	697	453
158	10000	4068	6088	47906	46406	6102	4085
393	25000	11003	15403	119342	117859	15418	11018

Tableau 8- 7: Comparaison de Temps de sauvegarde en utilisant les signatures algébriques et les signatures cryptographiques SHA-1

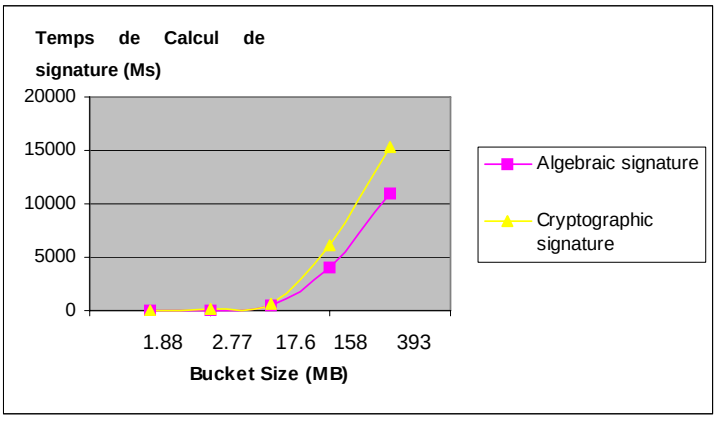


Figure 8- 4: Comparaison entre les signatures algébriques et les signatures cryptographiques SHA-1

D'après la figure 8-4, le gain par nos signatures algébriques augmente au fur et à mesure que la taille des enregistrements augmente (case de données de plus grande taille). L'utilisation de signatures algébriques est plus bénéfique pour les grandes cases de données.La taille de 20 *Octets* pour chaque signature cryptographique est coûteux ce qui constitue un avantage majeur pour les signatures algébriques.

8.4 Restauration d'un Fichier SDDS

Après une sauvegarde d'un fichier sur disque dur, un client doit pouvoir lancer la restauration de ce fichier vers la mémoire centrale des différents serveurs concernés. Nous effectuons des expérimentations sur un fichier déja sauvegardé sur disque.

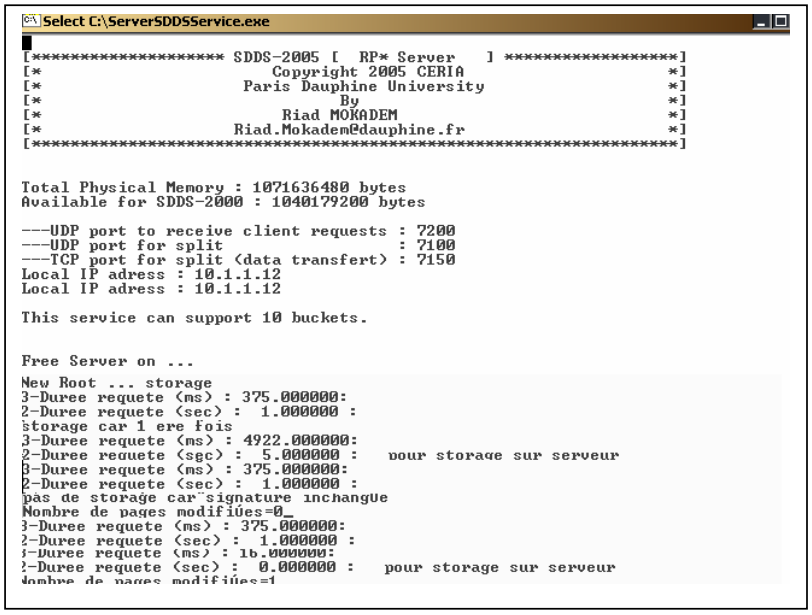
Pour les même raisons que lors d'une sauvegarde d'un fichier, le chargement d'un fichier distribué sur plusieurs serveurs prend moins de temps que son chargement sur un disque local. Cela est possible grâce au parallélisme offert par les systèmes SDDS [M02]. Les temps de chargement d'un fichier SDDS sont inférieurs à ceux de sa sauvegarde sur disque. A chaque rajour d'un serveur, le temps de chargement de fichier diminue de 7%. Ainsi, le chargement d'un fichier distribué sur 3 serveurs montre une économie de 22% par rapport au chargement de ce meme fichier sur un seul serveur.

De même, si nous simulons une panne d'un serveur (défaillance d'une case de données) ou même plusieurs de ces serveurs, nous pouvons récupérer la totalité des données se trouvant sur chacun de ces serveurs. Néanmoins, c'est les données sauvées lors de la dernière sauvegarde qui seront chargées. Dans le domaine de récupération de données, une perspective consiste à comparer l'utilité de nos signatures algébriques avec les travaux utilisant les *données de parités*. Ces données sont utilisées au niveau de ses serveurs pour la récupération d'un enregistrement ou d'une case lors d'une défaillance d'un serveur. Elles se basent sur les codes de *Reed-Solomon* [M04]. Cela va dans le même sens que la restauration d'un fichier sauvegardé précédemment pour remédier à un éventuel crash (accident) ou pour libérer de la mémoire. Bien que les données de parité soient en mémoire, le temps du chargement d'un fichier est très réduit dans le cas répartition de ce dernier sur plusieurs serveurs. Dans notre cas, nous n'avons pas à reconstituer les données de la case nécessitant des milliers d'itérations. En plus, notre application permet de récupérer les données sur les disques des serveurs quel que soit le nombre de serveurs en panne.

8.5 Exemple de Sauvegarde de Fichier SDDS (Interface Serveur de Données)

La Figure 8- 5 montre un exemple d'interface au niveau du serveur de données. Après la création d'un fichier de données. Le client lance une opération de sauvegarde sur le serveur après avoir inséré 100 enregistrements.

A la première requête de sauvegarde, le serveur met la totalité du fichier sur disque après calcul de signatures. Si le client envoi une autre requête de stockage, le serveur de données ne sauvegarde aucune page. On prélève un gain de prés de 4922 ms. Supposons à présent, que le client effectue une insertion d'un seul enregistrement dans le même fichier. Lors de la prochaine requête de stockage, le serveur de données procède au calcul de signatures algébriques. Une comparaison est faite avec l'ensemble de signatures algébriques présentes en mémoire. Cela conduira à la détection d'une seule page de données mise à jour. Le temps nécessaire à cette requête de stockage sera de 375 + 16ms. Le gain est évalué à 99% du temps de stockage global.



1 ére Requete de Sauvegarde
Calcul de Signatures 375Ms
Stockage sur Disque
4922Ms

Sauvegarde sans Mise à jour : 375 Ms

Mise à jour 1 seule page
Sauvegarde : (375+16) Ms

Figure 8- 5 : Exemple de traitement requêtes de Sauvegarde au niveau d'un serveur.

8.6 Tables de Horner et de Multiplication par α

Nous avons étudié l'utilisation de tables de multiplication prés calculés. Cela est possible grâce à l'implémentation d'une table de multiplication par α^j. De meilleurs résultats peuvent etre obtenus si

l'ensemble des tables *Mul1*, *Mul2* et *Mul3*… (Chapitre 4) réside en cache L1 par exemple. L'utilisation de schéma de Horner est aussi bénéfique en utilisant les signatures inversées. Elles peuvent être efficace lors de l'encodage de données. La recherche de chaines est par contre plus complexe en utilisant ce type de signatures. Nous n'avons pas implémenté les signatures inversées dans notre système.

Une autre méthode consiste à l'utilisation de table de Broader. Ici, on présente des mesures pour le calcul de signatures algébriques pour chacune de ces techniques. Ces expériences ont été faites avec la collaboration de l'université Santa Clara. Des comparaisons sont faites. Les résultats sont dans le tableau qui suit :

Taille de la signature (Octets)	Longueur des caractères (Octets)	Type de multiplication utilisé pour le calcul de signatures	Temps de calcul de signatures (sec)
4	8	Tables *Log* et *Antilog* standard	0.523
4	8	Tables *multiplication* par α, α^2, α^3	0.325
4	8	Table *multiplication par* α	0.385
4	8	Table *Broader*	0.497

Tableau 8- 8: Comparaison de Temps de calcul des signatures algébriques suivant différents méthodes.

D'après ce tableau, il apparaît clairement que l'utilisation de tables pré calculées de multiplication par α, produit les meilleurs résultats. Son implémentation constitue une bonne initiative pour les travaux futurs.

8.7 Recherche de Chaîne par la Méthode '*XOR*'

La recherche par le contenu est une nouvelle primitive implémentée dans SDDS-2005. Elle concerne la partie non clé de chaque enregistrement. On effectue des mesures en utilisant la méthode *XOR* décrite dans le chapitre 7. Celle ci constitue une alternative pour la recherche de chaines en utilisant l'opération *XOR* ainsi que les signatures algébriques dans GF (2^{16}).

En utilisant a commande d'insertion par intervalle, nous avons varié le nombre d'enregistrements insérés à chaque expérience. Ici, chaque enregistrement a une taille de 24 *Octets*. Nous augmentons cette taille à 60 *Octets* pour d'autres expériences. Ensuite, nous insérons un dernier enregistrement contenant la chaîne que nous recherchons. La taille de ce dernier enregistrement est de 20 *Octets*. Puis, nous recherchons une chaîne de 6 *Octets*. Nous nous intéressons à deux types de recherches. D'abord, la *recherche complète*. Dans ce cas, l'enregistrement de taille 25 Octets sélectionné contient la totalité de la

chaîne, de taille *n* également, envoyée aux serveurs de données. Puis, nous expérimentons avec la *recherche partielle*. Dans ce type de recherche, la chaîne recherchée, de taille *m*, fait partie du contenu de l'enregistrement, de taille *n*, sélectionné tels que *m*<*n*.. Pour les expériences de recherche partielles que nous avons effectués, deux types d'expériences ont été faites. D'abord, la valeur *XOR* de la chaine recherchée est calculée au niveaun du client.puis envoyée au serveur. La comparaison concerne d'abord les valeurs '*XOR*' au sein de l'enregistrement puis des signatures algébriques en cas d'égalité. Les temps de recherche par cette méthode sont dans la colonne 3 et 5 du tableau 8-10.

Ensuite, dans la colonne 4 et 6 du meme tableau, une comparaison directe des signatures algébriques est faite sans passer par la 'valeur XOR'. Le Tableau 8- 9 montre les résultats obtenus pour la recherche d'une chaîne de 10 *Octets* dans une case contenant *n* enregistrements de taille *T* égale à 25 Octets. Cette taille sera de 60 *Octets* dans d'autres expériences.

Position de la chaîne dans la case (n)	Temps de recherche complète (ms) *T=25 Octets*	Temps de recherche Partielle (first *Xor* +sign if equal) (ms) *T=25 Octets*	Temps de recherche Partielle (Sign alg seules) (ms) *T=25 Octets*	Temps de recherche Partielle (first *Xo*r +sign if equal) (ms) *T=60 Octets*	Temps de recherche Partielle (Sign alg seules) (ms) *T=60 Octets*
99	4.26	8.8	9.15	17.4	17.42
199	8.58	18.32	19.005	34,537	34.8
299	13.4	27.2	28	52.4	52.49
499	23.9	47.92	49.05	90.37	89.54
990	49.1	100.8	101.1	182.3	184.189
3990	202	410.3	412.1	742.2	750.2
7990	410	840.5	844.3	1489.6	1501.1

Tableau 8- 9 : Temps de recherche complète et comparaison entre les 2 méthodes pour la recherche partielle

La recherche de chaîne complète est plus ou moins rapide vu qu'au niveau de chaque enregistrement, une seule comparaison est faite. Si la chaîne est trouvée au niveau de *i* ème enregistrement, la complexité de recherche est alors *O (i)*. Par ailleurs, Si on varie la taille des enregistrements, le temps de recherche reste le même vu qu'il y'aura toujours une comparaison dans l'en-tete de l'enregistrement (1 seule opération par enregistrement). De même, si la taille de la chaîne à rechercher varie, le temps de recherche reste inchangé pour la même raison [ML06].

Pour la recherche partielle, d'après le tableau, il est clair que les recherches utilisant les comparaisons de signatures algébriques uniquement nécessitent un peu plus de temps que celles utilisant d'abord des comparaisons de valeurs '*XOR*'.

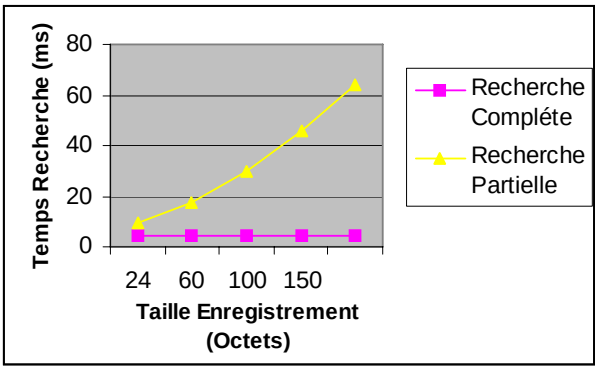


Figure 8- 6 : Temps de Recherche d'une chaîne fixe (10 octets) trouvée au 100ème enregistrement.

Dans d'autres expériences, nous varions la taille des enregistrements insérés. La Figure 8- 6 montre un temps inchangé pour la recherche complète L'essentiel du temps de la recherche complète est mis lors du passage d'un enregistrement à l'autre ce qui explique que ce temps est relativement grand (une seule comparaison pour la recherche complète au lieu de 15 pour la recherche partielle dans la troisièmee colonne pour la première expérience).

Pour la recherche partielle par la méthode Xor, le temps de recherche augmente au fur et à mesure que la taille d'enregistrements augmente. Le tableau précédent montre que le calcul des valeurs 'XOR' est plus rapide que le calcul de signatures algébriques. Ce dernier nécessite plus d'opérations et d'accès aux tables *Log* et *Antilog*. Ce calcul ne s'effectue d'ailleurs qu'en cas d'égalités au niveau des valeurs '*XOR*' entre la chaîne reçue et le contenu de l'enregistrement. L'utilisation de comparaisons par les valeurs 'XOR' d'abord est alors bénéfique. Ce gain de temps est surtout visible lorsqu'il s'agit d'enregistrements de grandes tailles ou encore lorsqu'il s'agit de larges cases de données. Ceci résulte de l'augmentation du nombre de tests et de comparaisons.

Le temps de recherche partielle augmente si la taille de la chaîne partielle est plus petite (beaucoup) par rapport à la taille des enregistrements. Cela, à cause de la multiplication de nombre de tests et comparaisons dans l'enregistrement. Dans le tableau précédent, la recherche d'une chaîne de *10 Octets* dans un enregistrement de *25 Octets* nécessite 15 comparaisons au plus. Pour un enregistrement de 60 *Octets*, cela nécessite 60-10=50 comparaisons.

Dans une autre expérience (Figure 8- 7), nous varions la taille de la chaîne à rechercher. Nos enregistrements sont d'une taille fixe (ici 100 *Octets*). La chaîne à recherchée se trouve au dernier

enregistrement (dans une case de capacité de 100 Octets). Il aparait clairement que plus la taille de la chaîne à rechercher grandit, plus le nombre de comparaison décroît et donc le temps de recherche décroît.

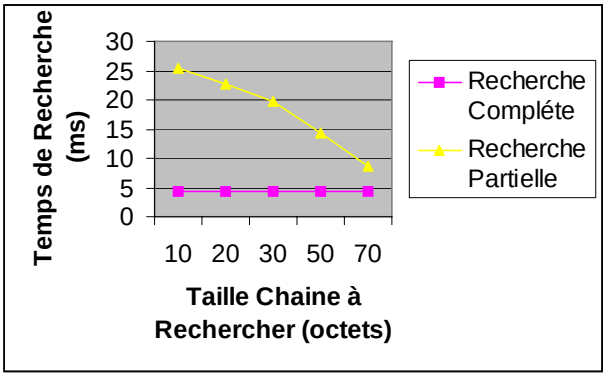


Figure 8- 7: Temps de Recherche suivant la taille de la chaîne à rechercher.

Rappelons que lors de nos expériences, nous mettons toujours la chaîne aboutissant au succès de la recherche, à la fin du dernier enregistrement. Ceci oblige le serveur au parcourt du contenu de tous les enregistrements. Ce parcourt concerne la partie non clé des enregistrement (ici 20, 60 ou 100 Octets). Le temps nécessaire pour retrouver les 10 caractères placés dans le dernier enregistrement est de 1.5 *sec* pour une case de 8000 enregistrements. Cependant, la traversée de la case (l'ensemble de passages d'un enregistrement à l'autre) nécessite 0.5 *sec*. En prenant en compte ce temps, la recherche nécessite en moyenne *2 sec* pour 1*MB*.

8.7.1 Comparaison de la Méthode *XOR* avec l'Algorithme de Karp-Rabin

Afin de valider les résultats de nos travaux, nous comparons la technique de recherche par la méthode *XOR* à d'autres schémas. L'un des plus connu dans le domaine de recherche de chaîne est l'algorithme de Karp-Rabin. Nous implémentons ainsi cet algorithme suivant le pseudo code [I04].

Dans le tableau 8-11, nous insérons des enregistrements de 950 *Octets* chacun contenant des caractères ASCII. Nous mettons dans le dernier enregistrement une chaîne de 30 *Octets*. Puis, nous recherchons une chaîne de 20 *Octets*. L'algorithme de Karp-Rabin lors de la recherche dans chaque enregistrement. Rappelons que chaque signature à une taille de 4 *Octets*. Le calcul se fait dans domaine GF (2^{16}).

Capacité de la case	Position de la chaîne à rechercher	Taille de la chaîne à rechercher	Temps de recherche partielle (ms)	Temps de recherche par Karp-Rabin (ms)
100	99	20	625	557
100	99	40	492	560
500	299	20	3552	2880
500	299	40	2806	2987

Tableau 8- 10 : Tableau de comparaison temps de recherche en utilisant Karp-Rabin / méthode XOR+ sign algébriques

L'algorithme de Karp-Rabin [KR87] utilise juste *deux multiplications* et une *addition* pour le calcul de valeurs de hachage. A cause de sa linéarité, il constitue un moyen très rapide pour la recherche de chaînes. Cependant, cela n'est valable que pour les chaînes de taille inférieure à 32 *Octets*. Comme nous pouvons le constater (Tableau 8- 10 et Figure 8- 8), il est plus efficace pour la recherche de petites chaînes qui ne dépassent pas 32 *Octets* et présente un gain non négligeable concernant le temps de recherche. Nos signatures algébriques, bien que moins rapide que Karp-Rabin, présentent l'avantage que la taille de la chaîne à rechercher n'est pas limitée.

Dans notre technique de recherche utilisant les signatures algébriques, le principal du temps de recherche est perdu lors des transferts de données en mémoire et du passage d'un enregistrement à l'autre. Les opérations arithmétiques dans la structure de corps de Galois étant très rapides. D'ailleurs, la moyenne de calcul de ces signatures n'excédant pas 5 *usec/KB* de données [LS04]. Pour une recherche d'une chaîne de 6 *Octets* dans 8000 enregistrements de 60 Octets chacun, le temps de recherche avec l'algorithme de Karp-Rabin est de 1.504 *sec* quand nos signatures aboutissent à un succès au bout de 1.516 *sec*. Lorsque la chaîne à rechercher devient plus importante (plus de 32 Octets), la technique de recherche par les signatures algébriques est plus rapide. La 2 ème et 4 ème ligne dans le tableau 9-11 montre le gain engendré. Dans notre cas, cette recherche concerne une chaîne de 40 *Octets*.

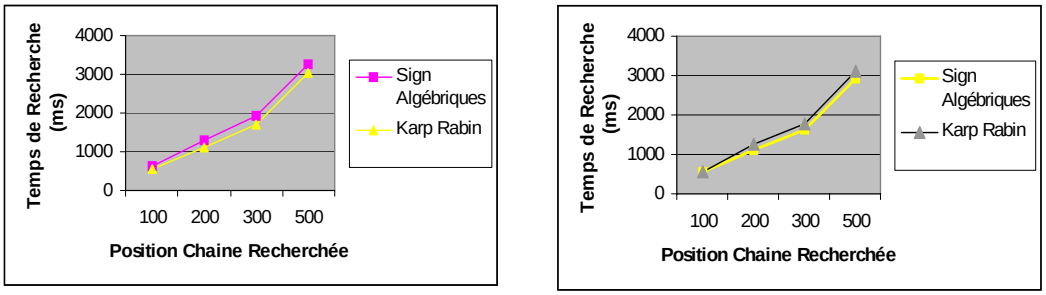


Figure 8- 8 : Comparaison entre temps de recherche d'une chaîne <32 Octets (à gauche) et >32 Octets (à droite) par les signatures Algébriques et l'algorithme Karp-Rabin

8.8 Mise à jour de Données en utilisant les Signatures Algébriques

Comme énoncé lors du chapitre 6, nous utilisons nos signatures algébriques pour éviter l'envoi d'enregistrements ayant été mis à jour avec une meme valeur. Ce genre de mise à jour est appelée « *pseudo mise jour* » [LS04, H03]. Nous testons nos signatures pour les 2 types de mises à jour : *normales* et *aveugles* (chapitre 4). Rappelons que dans la mise à jour normale, le client s'assure d'abord de l'existence de l'enregistrement à modifier contrairement au cas de mise à jour aveugle ou le client procède à une mise à jour d'un enregistrement dont l'existence n'est pas garantie.

Une mise à jour normale, suite à une opération de recherche, nécessite 0.92 *ms* pour un enregistrement de 1Ko. Sans changement dans cet enregistrement, cette mise à jour ne nécessite que 0.28 *ms* pour un même enregistrement ce qui équivaut au temps de la recherche de cet enregistrement [ML06]. Dans ce dernier cas, la comparaison au niveau du client indiquera une similarité de signatures entre l'*image avant* et l'*image après*. Le client ne procédant pas alors à la modification de l'enregistrement puisqu'il s'agit de mettre à jour son contenu avec une même valeur. Grâce à la comparaison de signatures, le gain ici est de 70%.

Une mise à jour aveugle nécessite 0.75 *ms* avec changement de données. Cela nécessite 0.20*ms* pour une mise à jour sans modification de données pour un enregistrement de *1Ko*. Le gain est aussi de 73%. La différence de temps avec une mise à jour normale est surtout due à l'absence de la recherche de enregistrement sur les serveurs de données.

Le tableau Tableau 8- 11 récapitule ces temps de recherche pour chacune des deux techniques :

Types de requêtes	Temps avec modification de données	Temps sans modification de données
Recherche de l'enregistrement lors de la mise à jour normale	0.27	---
Mise à jour normale (incluant le temps de recherche)	0.92	0.28
Mise à jour aveugle	0.74	0.20

Tableau 8- 11: Comparaison entre Temps de différents types de recherches.

Notons que ces temps incluent le temps d'accès à l'enregistrement, le temps de recherche de l'enregistrement (pour le cas d'une mise à jour normale), le calcul et comparaisons de signatures et le transfert de la signature de cet enregistrement. Pour des enregistrements de taille plus importante, le gain pour les « *pseudo* » mises à jour est d'environs 50%.

8.9 La Recherche de Chaînes dans SDDS-2005 (Signature Algébriques Cumulatives)

Vous valider nos travaux concernant les signatures cumulatives, nous effectuons diverses expériences relatives aux opérations d'encodage, de décodage et aux différents types de recherche permises dans SDDS-2005. Notons qu'ici que nos calculs de signatures sont fait dans *GF* (2^8). Les données insérées ainsi que la chaîne à rechercher ont une taille d'un octet.

Dans un premier temps, nous avons procédé à l'encodage des données avec des signatures algébriques prés calculés (non cumulatives). Chaque élément est remplacé par sa signature algébrique individuelle pré calculée. Elles sont calculées suivant la position de l'élément dans l'enregistrement. Néanmoins, l'introduction de signatures cumulatives a permis un gain un gain d'environs 25% lors des de la recherche partielle de chaines. Cela est possible grâce à la réduction du nombre d'opérations pour le calcul de la signature d'une chaîne de caractères.

Nous adoptons pour la suite de nos mesures, les signatures cumulatives. Nous opterons alors pour le remplacement de chaque élément individuel p par sa signature algébrique cumulative p'. Dans les éxpériences qui suivent, le client procède à l'insertion de chaînes de caractères de valeurs aléatoires dans ASCII de telle sorte que les enregistrements n'auront pas de mêmes valeurs répétitives dans leurs contenus.

8.9.1 Encodage / Décodage de Données

L'encodage n'est effectué qu'au niveau du client à l'insertion ou lors d'une mise à jour d'un enregistrement. Le processus de décodage est nécesaire lors de la réception du résultat d'une recherche par exemple (au client). Voyons le coût de cet encodage/ décodage à travers le Tableau 8- 12 montrant l'insertion d'un enregistrement, son encodage, sa recherche et son décodage.

Taille de l'enregistrement (Octets)	Temps d'insertion (ms)	Temps de recherche par clé (ms)	Temps d'encodage (ms)	Temps de décodage (ms)
1000	0.27	0.28	0.044	0.041

Tableau 8- 12: Temps d'encodage / décodage et comparaison avec d'autres opérations.

Le processus d'encodage/ décodage est d'autant avantageux que le temps nécessaire pour accomplir cela est très réduit. Ainsi, pour encoder un enregistrement de 1*Mo*, cela nécessite 43 *ms*. En moyenne, l'encodage est de 0.043 *ms*/ *Ko*. Rappelons qu'une insertion d'un enregistrement de même taille nécessite 0.26 *ms* [DL01] ce qui rend le temps d'encodage assez petit.

L'opération de décodage nécessite un peu moins de temps 0.041 *ms*/ *Ko*. En comparaison au temps d'insertion et de recherche en général, l'encodage/ décodage nécessite entre 16% et 14% de ces temps respectivement (Figure 8- 9). Ce temps englobe l'accès aux tables Log et Antilog nécessaires pour le calcul des signatures ainsi que quelques opérations *XOR*.

De plus, le processus c'encodage / décodage ne nécessitant aucune ressource de stockage en mémoire comme c'est le cas lors des techniques de cryptage de données.

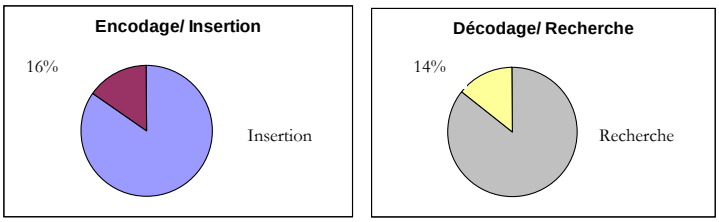


Figure 8- 9 : Comparaison des temps d'encodage et de décodage / temps d'insertion et de recherche (resp)

8.9.2 Recherche de Préfixes et de Chaînes de Caractères

Dans ce type de recherche, les signatures sont calculées d'abord au niveau du client. Il n'y a pas de transfert de chaînes de caractères vers les serveurs. Le client envoi juste la signature du préfixe (ou de la chaîne) à rechercher accompagnée de la taille de ce préfixe ou de la chaîne à rechercher.

Pour certaines expériences, nous insérons la chaîne à rechercher à la fin du premier enregistrement (1ère ligne du Tableau 8- 13). Dans d'autres expériences, cette chaîne est insérée à la fin du dernier enregistrement de la case obligeant le serveur à parcourir l'ensemble des enregistrements.

Position préfixe dans case	Taille des enreg insérés	Taille du préfixe à rechercher	Temps de recherche (ms)
1	100	20	0.369
100	250	20	37.8
100	250	35	37.7
200	250	20	71.3
300	250	20	120.53
500	250	20	197.5

Position chaîne dans case	Taille des enreg insérés	Taille chaîne à rechercher	Offset dans enreg	Temps de recherche (ms)
1	20	5	13	0.44
1	100	20	70	0.68
1	100	20	80	0.689
100	250	15	80	451
100	250	30	80	437
200	250	15	80	884

Tableau 8- 13: Temps de Recherche de préfixe (à gauche) et de chaîne de caractères (à droite).

La recherche du préfixe situé au 1er enregistrement de taille 100 *Octets* nécessite 0.37*ms*. Ce temps est très intéressant vu que l'insertion d'un enregistrement de 100 *Octets* avoisine 0.27*ms*. Ici la taille du préfixe à rechercher est de 20 *Octets*. Mais, la taille ici n'est pas très importante vu que l'algorithme procède par un seul accès avec une complexité de recherche de *O (1)* pour chaque enregistrement. Le temps mis pour rechercher un préfixe de 100 *Octets* serait alors le même On peut voir cela à travers la 2ème et 3 ème ligne du tableau 8-14 (à gauche). Cela n'est pas le cas avec l'utilisation de la méthode *XOR* qui nécessite le passage par *m* caractères pour chaque enregistrement (*m* étant la taille du préfixe recherché). De plus, la recherche par clé d'un même enregistrement avoisine 0.3 *ms*. On pourra dire que la recherche du préfixe offre un temps très intéressant. Cette recherche est linéaire pour *n* enregistrements. Le Tableau 8- 13 (g) montre une linéarité pour la recherche de préfixe se trouvant à la 100 ème, 200 ème position.

Pour la recherche de chaîne de caractère, le temps nécessaire pour trouver une chaîne de 20 *Octets* est de 0.68*ms*. Cette chaîne étant situé à la fin du le 1er enregistrement. Cela a nécessité 80 comparaisons et le temps de complexité ici est de *O (n-m)* au pire des cas. Nous avons répété nos expériences en variant la taille de la chaîne à rechercher (5 Octets puis 20 Octets) et la taille des enregistrements (100 puis 250 Octets).

On remarque que l'offset de position de la chaîne à rechercher dans l'enregistrement est important par rapport au nombre de tests (ligne 2 et 3 du Tableau 8- 13 à droite). La taille de la chaîne à rechercher est aussi importante. Quand celle ci est trop petite par rapport à la taille de l'enregistrement parcouru (surtout si elle est positionné à la fin), il y'a plus de comparaisons. Le temps de recherche deviens alors plus important (4 ème et 5 ème ligne du tableau à droite ci dessus).

Suivant les deux tableaux en comparant avec les résultats obtenus avec l'utilisation des signatures algébriques (tableau 8-14), un gain de 30% est observé pour nos signatures algébriques cumulatives. Ce gain devient plus important lorsqu'il s'agit de larges cases de données.

8.9.3 Comparaisons de nos Travaux avec l'Algorithme Karp-Rabin

Nous comparons les mesures précédentes avec ceux obtenus avec notre implémentation de l'algorithme de Karp-Rabin [I04], sans doute le plus efficace et rapide dans le domaine de la recherche de chaînes de caractères (d'autres algorithmes existent, l'algorithme de Boyer-Moore, Knuth-Morris-Pratt et celui surnommé Quick search le plus rapide pour la recherche de toutes petites chaînes).

Le tableau suivant montre une comparaison entre la recherche de chaines par les signatures cumulatives et celle utilisant la méthode Xor dans Gf (2^{16}). Nous comparons ensuite ces résultats à la recherche par l'algorithme de Karp-Rabin :

Position dans la case	Taille des enregist-rements	Taille du dernier enreg	Taille chaîne à re-chercher	Offset dans dernier enreg	Temps de recherche (XOR+Alg sign) (ms)	Temps de recherche (karpRabin) (ms)	Temps de recherche (sign cumulatives) (ms)
100	200	25	10	5	205	151	147
200	200	25	10	5	368	275	268
500	200	25	10	5	1123	725	716
100	200	45	35	5	182	168	126

Tableau 8- 14 : Tableau comparatif entre le temps de recherche en utilisant les signatures cumulative et l'algorithme Karp-Rabin

Notre implémentation par les signatures cumulatives présente un gain de *5%* par rapport aux résultats obtenus avec l'algorithme de Karp-Rabin (les premières lignes du tableau). Ce gain est assez petit sans doute à cause de la linéarité dans le calcul des valeurs et l'utilisation de moindre opérations tels que opérations de '*XOR*' dans Karp-Rabin.

Dans l'algorithme KR, la taille de la chaîne à rechercher est limitée à 32 *Octets*. Ceci n'est pas le cas lors des recherches utilisant les signatures algébriques cumulatives. En effet, dés qu'on dépasse la taille de *32 Octets* pour la taille de la chaîne à rechercher (Figure 8- 10), notre algorithme de recherche par signatures cumulatives devient encore plus rapide et offre un gain pouvant atteindre 25%. Ici, on utilise

une chaîne de *35 Octets* dans la dernière ligne du tableau. Ce gain augmente au fur et à mesure qu'on augmente la taille de notre chaîne recherchée.

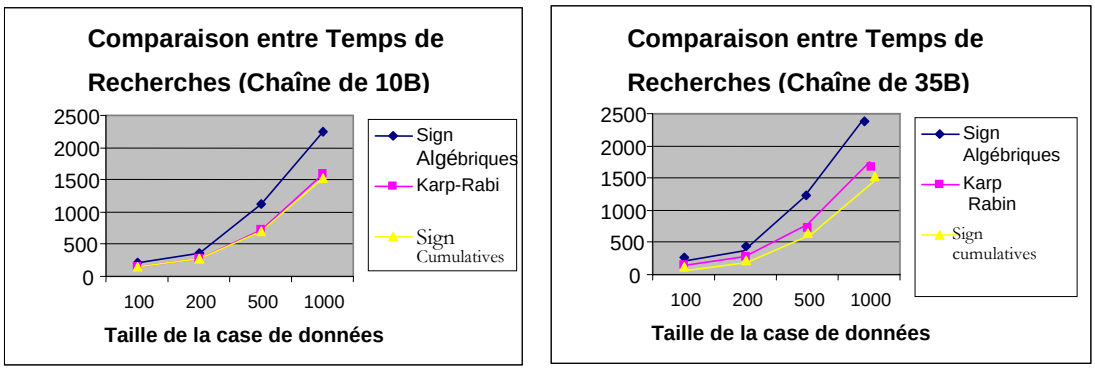


Figure 8- 10 : Comparaison entre temps de recherche d'une chaîne <32B (à gauche) et >32B (à droite) par les signatures cumulatives et l'algorithme Karp-Rabin

Cependant, nos signatures cumulatives imposent également une taille limite de 256 Octets pour cette chaîne à rechercher due à l'implémentation dans GF (2^8). Nous remédions à cela en utilisant le *Modulo [255]* dans notre implémentation. Ceci est fait pour ne pas provoquer de débordement dans la table Antilog.

Divers solutions existent aussi pour remédier à cette limite. D'autres techniques permettent de dépasser la taille de 256 Octets, on citera l'implémentation de table *Antilog* plus grande ou encore de tables pré calculées de *multiplication par alpha*. Nous n'avons implémenté que celle utilisant le *modulo*. Des expériences ont montré que l'introduction de ce 'M*odulo*' n'augmente que de 1% à 2% le temps de nos recherches ce qui constitue un avantage majeur pour nos signatures cumulatives dans le cas de recherche de longues chaînes de caractères. De plus, cela n'est pas aussi 'dangereux' du moment qu'on procède par la résolution de collision à la fin de chaque recherche.

8.9.4 Recherche du Plus Grand Préfixe

Dans ce type de recherche, le client envoie la chaîne de caractère représentant ce préfixe. La taille de ce préfixe est aussi envoyée. Les différentes signatures sont calculées au niveau des serveurs de données. Nous effectuons différentes expériences en variant le positionnement du plus grand préfixe dans nos enregistrements. Durant ces expériences, le préfixe envoyé par le client est de taille 25. Le plus grand préfixe commun trouvé étant de 20. Le tableau qui suit montre les résultats obtenus

Position plus grande préfixe dans case	Taille des enreg insérées	Taille dernier enreg (Octets)	Taille préfixe à rechercher	Temps de recherche (ms)
1	50	50	25	0.38
1/100	250	50	25	43
50/100	250	50	25	46.2
99/100	250	50	25	53.2

Tableau 8- 15 : Temps de recherche du plus grand préfixe

Dans la dernière expérience (4 ème ligne du Tableau 8- 15), les 98 enregistrement comportent des préfixes intermédiaires mais de tailles inférieure au préfixe inséré dans le dernier enregistrement. Avant d'atteindre le dernier enregistrement, une opération au minimum (cas d'échec) est effectuée pour chaque enregistrement. Le nombre d'opérations augmente surtout si des enregistrements presque similaires sont rencontrés. Le temps nécessaire pour trouver le plus grand préfixe commun se trouvant au 99 ème enregistrement, est maximum.

Notons qu'une fois un préfixe commun est trouvé avec une taille *L*, la recherche (et donc la comparaison) au prochain enregistrement commencera à la position *L+1*.

Dans le cas 51/ 100, l'enregistrement contenant le préfixe recherché se trouve à la position 51. Le serveur parcourt les 50 premiers enregistrements en faisant plus d'une comparaison sur certains de ces enregistrements. A partir du 51 ème, une seule comparaison est faite aboutissant à un échec à chaque fois jusqu'à atteindre le dernier enregistrement de la case. Le temps nécessaire est inférieur au temps précédent. Les signatures algébriques cumulatives sont d'une importance majeure ici du fait qu'une seule comparaison est faite à la position supérieure à celle du plus grand préfixe actuel. Cette seule comparaison permettra de se fixer sur le succès ou l'échec de trouver un nouveau plus grand préfixe commun.

8.9.5 Recherche de la Plus Grande Chaîne Commune

Pour ce type de recherche, le client envoie la chaîne recherchée aux serveurs de données. Dans ces expériences, nous insérons *n* enregistrements. Pour certain de ces enregistrements, ils contiennent une partie de la chaîne envoyée par le client. Dans l'ensemble des enregistrements de la case, nous nous intéressons à la plus grande chaîne commune avec la chaîne reçue. A chaque expérience, nous nous arrangeons pour mettre cette chaîne commune dans le dernier enregistrement de la case. Cette chaîne est aussi mise à la fin de cet enregistrement. Supposons une chaîne *T* envoyée par le client. Pour

chaque caractère de la chaîne reçue, un parcourt de l'enregistrement actuel est fait à partir du début. Le parcourt au sein de l'enregistrement se fait suivant des puissance de 2.
Le Tableau 8- 16 montre des résultats de mesures de recherche de plus grande chaîne commune suivant l'algorithme discuté dans la section 7.3.4.

Position plus grande chaîne commune dans case	Taille des enreg insérées	Taille dernier enreg	Taille chaîne à rechercher	Offset chaîne dans dernier enreg	Temps de recherche (ms)
1	100	100	25	70	0.68
100	100	45	10	10	290
100	100	45	15	10	470
100	120	45	15	10	565

Tableau 8- 16 : Temps de recherche de la plus grande chaîne commune.

Pour chaque caractère de la chaîne recherchée, un parcourt est fait dans tous l'enregistrement. A cause de toutes ces comparaisons pour un même enregistrement, le temps pour retrouver la plus grande chaîne commune est bien plus supérieur à la recherche d'une chaîne ou encore du préfixe.

Le temps nécessaire pour trouver une chaîne commune de 25 *Octets* situé à la fin d'un enregistrement de 100 *Octets* est de 0.68 *ms*. Ce temps est beaucoup plus grand lorsqu'on augmente la taille des enregistrements (100 *puis* 120 *Octets*) ou lors du parcourt de centaines d'enregistrements. L'augmentation de la taille de chaîne envoyée par le client augmente aussi ce temps du fait du parcourt séquentiel dans la chaîne (caractère par caractère). L'augmentation de la taille de la chaine envoyée augmente également ce temps de recherche.

8.9.6 Recherche sur Plusieurs Serveurs

Nous voulons tester la scalabilité de notre schéma de recherche. On insère 99 enregistrements dans chacun des *3* serveurs de données puis nous mettons un préfixe commun au dernier enregistrement de chaque serveur de données. Le client procède par la suite à l'envoi aux serveurs d'une chaîne contenant un préfixe (requête de recherche du plus grand préfixe). Chacun des serveurs recherche le plus grand préfixe commun à celui reçu dans la case de données correspondante. A la fin d'opération sur chaque serveur, celui ci procède à la résolution de collision (stratégie de résolution de collision au chapitre 7) puis à l'envoi du résultat au client dans un message contenant : la clé de l'enregistrement sélectionné et la taille du préfixe le pus long.

En comparant nos résultats, on constate que le temps de recherche sur plusieurs serveurs est nettement inférieur à celui s'effectuant sur un serveur de données contenant 300 enregistrements. Le temps de

recherche sur l'ensemble des serveurs contenant chacun 99 enregistrement est presque semblable (158*ms*) au temps de recherche sur un seul serveur contenant cette fois ci 99 enregistrements seulement (155*ms*) d'où l'importance de l'utilisation de structure de données distribuées générant un gain important au niveau de la recherche de chaînes de caractères La petite différence constatée est due à la comparaison faite au niveau du client pour le choix du plus grand préfixe dans les 3 serveurs de données. Cela répond à nos attentes ce qui confirme la scalabilité de notre schéma.

8.10 Exemple de Recherche par le contenu dans SDDS-2005

La Figure 8- 11 montre l'insertion d'un enregistrement de clé 123 contenant du texte XML (Copie d'écran à droite). Le client recherche par le contenu le texte 'death'. La partie droite de la figure montre les différenbts types de recherche par le contenu. Dans notre cas, c'est la commande N° 5 correspondante à la recherche par les signatures cumulatives qui est selectionnée. Le serveur trouve cette chaîne à l'enregistrement de clé 123 et le signale au client.

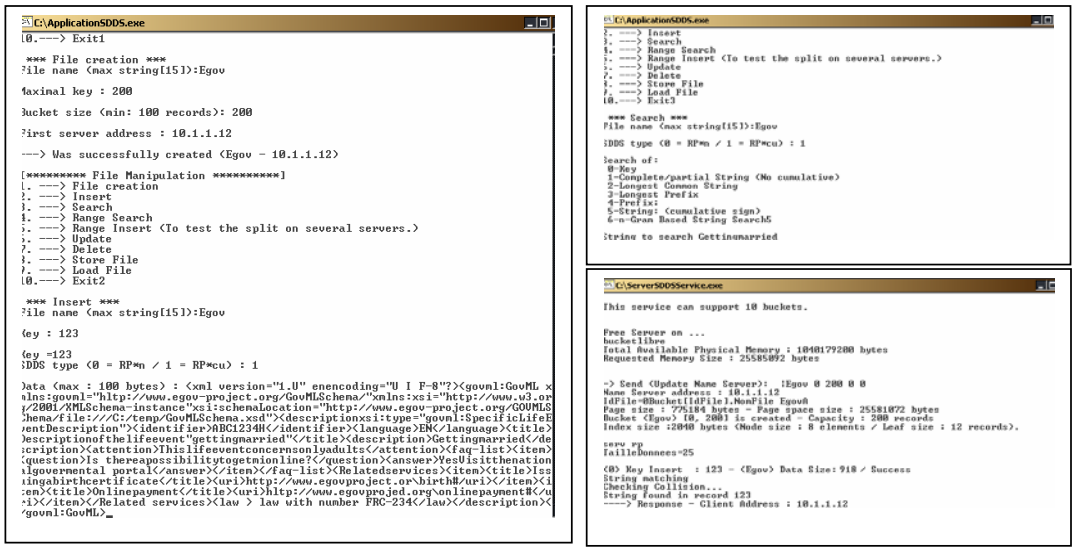


Figure 8- 11 : Copies d'écrans pour une insertion puis une recherche par le contenu.

8.11 Recherche de Chaînes par la Méthode *n*-Gramme

Nous implémentons l'algorithme de recherche par la méthode *n*-Gramme puis nous réalisons des expériences. Pour pouvoir comparer et analyser cette méthode, nous comparons les résultats obtenus avec ceux des mêmes expériences utilisant les signatures cumulatives.

Chaque client envoie la chaîne à rechercher aux serveurs de données. La table *n-Gramme,* construite au début du traitement, est calculée au niveau de chaque serveur concerné par la recherche. Nous insérons un seul enregistrement de taille *n* contenant une chaîne *M* de taille *k* que nous recherchons par la suite. Cette chaîne est censée être sélectionnée par le serveur de données lors du traitement de la recherche.

La taille de notre enregistrement est plus ou moins importante (*n*=300 Octets) pour mieux analyser les sauts surtout lorsqu'il s'agit de l'échec de la recherche sur plusieurs caractères dans cet enregistrement. Par la suite, nous varions la taille de cette chaîne pour mieux analyser les avantages de chacune des méthodes testées. Le choix de rechercher dans un seul enregistrement plutôt que dans l'ensemble d'une case, constituée de plusieurs enregistrements, est motivé par le fait que le passage d'un enregistrement à un autre (parcourt de l'index de l'arbre B) est relativement lent par rapport au temps de calcul des signatures cumulatives et traitement par la méthode n-Gramme.
Dans ces expériences, on fixe *n=2* (*digrammes*). Cette valeur est valable pour des données moins sélectives que celles utilisant *n=1*. Avec l'utilisation de digrammes, ce saut moyen devient *k-1* et la probabilité d'égalité sur les digrams avoisine ¼. Ceci réduit le nombre de succès par rapport à la recherche dans la table possible grâce à la réduction de nombre d'égalité sur les n-Grammes.Analysons à présents les mesures de temps de recherche obtenus par les deux méthodes et représentés dans le tableau suivant :

Chaîne recherchée	Temps de Recherche (n-Gramme) (us)	Temps de Recherche (Cumulatives) (us)	Gain
5	85	438	**5.15**
10	47	422	8.97
25	29	406	14
50	25	392	15.6
70	21	345	19.38
100	9	200	22.2
140	6	135	**22.5**

Tableau 8- 17 : Mesures de Temps de recherche par la méthode *n*-Gramme (Comparaison avec les signatures cumulatives)

Dans nos expériences, les différentes inégalités sur les digrams puis la non disponibilités de ces digrams (*n=2*) dans la table *T* conduit à une complexité de recherche de *O (M-k)/ (k-1)* qui constitue le coût de la recherche (notre cas). Cette complexité est de *O (M-k)* dans le cas de recherche par les signatures cumulatives. Une amélioration d'environs (*k-1*) fois est constatée par rapport au précédent algorithme, lui déjà, aussi rapide que les plus connus des algorithmes de recherche de chaînes existants (plus rapide pour la recherche du préfixe par exemple).

Les temps mesurés montrent que la méthode de recherche *n-Gramme* présente un gain important par rapport à la recherche utilisant les signatures cumulatives. Ce gain est déjà observé pour la recherche de petites chaînes (5 ou 10 *Octets*) et il est entre 5 et 8 fois pour toute chaîne nettement inférieure à la taille de l'enregistrement.

Si la taille de la chaîne à rechercher devient plus importante (100, 150 *Octets*), cet ordre d'amélioration devient plus important atteignant les 22 fois et même beaucoup plus si on fait d'autres mesures. Ce gain est observé quand la taille de la chaîne recherchée se rapproche du tiers ou la moitié de l'enregistrement.

Prenons par exemple le cas de la recherche d'une chaîne de 100 *Octets* dans un enregistrement de 300 *Octets*. Dans le cas de la recherche par les signatures cumulatives, prés de 200 comparaisons sont nécessaires (valeur maximale correspondante au cas du succès à la fin de l'enregistrement).

Avec la méthode *n-Gramme*, si les différents digrams testés ne sont pas présents (sinon vers la fin) de la *table digram*, le meilleur cas ne nécessite que 4 sauts. Pour chaque digramme absent dans la table digram, un saut de 99 caractères est effectué (en un seul coup) .

Prenons le cas de la première ligne du

Tableau 8- 17 Celui ci montre la recherche de *5* caractères dans un enregistrement comportant 300 caractères. Théoriquement, le gain de l'utilisation de notre méthode par rapport à celle utilisant les signatures cumulatives devrait être de l'ordre de 295/75=5.26. Dans notre expérience, on obtient un gain de 5.15 (valeurs proches).

Le nombre de tests nécessaire par la méthode de signatures est 295 (295=300-5). Il est de 75 (75=300/5) lors de la recherche parla méthode *n*-Gramme. Au fur et à mesures qu'on augmente la taille de la chaîne à rechercher, ce gain augmente également de façon exponentielle.

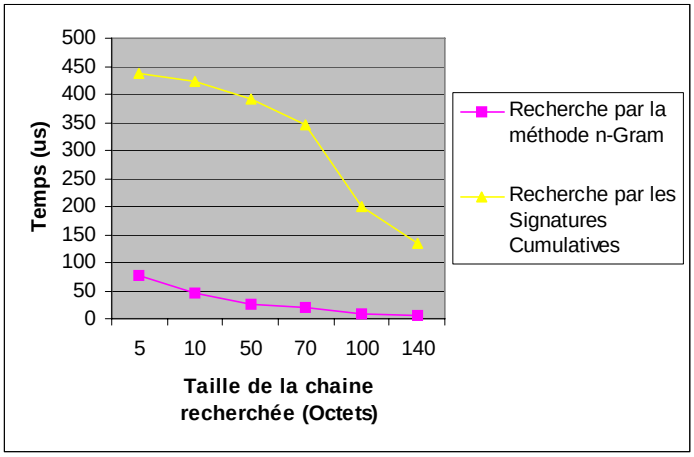


Figure 8- 12 : Comparaison entre la Recherche par la méthode *n*-Gram et les Signatures Algébriques et Cumulatives

L'analyse des résultats obtenus confirme nos attentes prévoyantes. Notre algorithme est encore meilleur lorsqu'il s'agit de recherche de longues chaînes de caractères Cela résulte des longs sauts, possibles grâce à l'utilisation de la table *n-Gramme*. En effet, ces sauts sont plus importants que ceux obtenus par la recherche utilisant les signatures cumulatives. La construction initiale de table n-Gramme et les comparaisons dans cette table sont négligeables par rapport au gain en nombre de sauts ce qui induit un gain de temps considérable.

En comparant les temps de recherche, utilisant les 2 méthodes (Figure 8- 12), le rapport entre la recherche augmente au fur et à mesure qu'on augmente la taille de la chaîne à rechercher. Ainsi, il passe de *5* fois pour une petite chaîne à *20* fois pour une chaîne de 140 Octets.

Cette recherche est non seulement valable pour le cas de recherche dans nos structure de données SDDS mais aussi lors d'une recherche de chaînes dans un fichier on une base de données…

Exemple 8.1

L'exemple de la Figure 8- 13 illustre un exemple de recherche pour une *séquence DNA*. Cet exemple est inspiré de [CL03]. Nous disposons d'un alphabet de nucléotides : A, C, G, T. La chaîne à rechercher ici est :AGACAGAT.

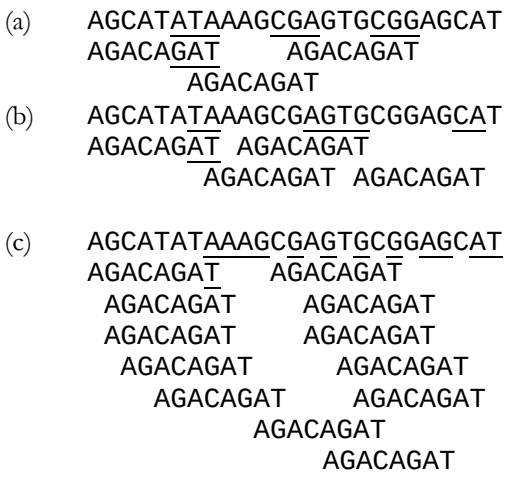


Figure 8- 13 : Recherche n-Gramme dans une séquence DNA.

Un choix de *n=1* nécessite *12* tests. Un choix *n=2* réduit ce nombre à *4* comparaisons. Le meilleur choix de n ici est de *n=3* qui ne nécessite que *3* tests. Cela accélère la recherche par un facteur de *4*. Un plus grand n ne donne pas de meilleurs résultats que *n=3*.

Notons que Boyer-Moore nécessite le même nombre de tests que pour *n=1* ce qui montre qu'il est plus lent que la recherche par trigramme *(n=3)*.

Cet algorithme semble particulièrement simple d'usage et rapide. Il est particulièrement efficace pour la recherche de longues chaînes telles que les images, les empreintes, les codes DNA…

Chapitre

9 Conclusion & Perspectives

Cette Thèse avait pour objectif la contribution à la réalisation et l'implémentation d'un gestionnaire de Structures de Données Distribuées et Scalables basé sur la famille des SDDSs RP* et opérationnel sur un multi-ordinateur réseau. Cela passe par l'implantation et la conception de diverses primitives et nouvelles fonctions. Plusieurs de celles ci ont été implémentées dans le prototype SDDS-2005. Elles sont destinées à la sauvegarde de données sur le disque, un traitement plus efficace de mises à jour, le traitement de concurrence ainsi que celui de la recherche par le contenu (scans). Enfin, pour mieux répondre au contexte P2P, il nous fallait introduire une certaine protection de données, au moins contre une découverte accidentelle de leurs valeurs. Ceci nous a conduit au problème intéressant de recherche de données par l'exploration directe de leur contenu encodé, sans décodage local.

Nous avons basé l'ensemble de nos fonctions sur une technique nouvelle dite de signatures algébriques. Celles-ci sont d'une grande contribution pour la gestion de ces données dans SDDS-2005. Elles sont calculées dans la structure de corps de Galois. Nous avons détaillé la théorie et notre pratique de signatures algébriques tout au long de cette Thèse. Ainsi, une sauvegarde sur disque n'écrit que les parties de la RAM modifiées depuis la dernière sauvegarde. Nous permettons aussi un accès concurrentiel aux données par les clients sans attente. Le contrôle de concurrence est optimiste, sans verrouillage, pour de meilleures performances d'accès. Aussi, les mises à jour ne sont effectuées que si

leurs contenus sont réellement modifiés et l'enregistrement mis à jour n'est alors envoyé au serveur que s'il est réellement modifié. Puis, les données stockées sont suffisamment encodées au client pour rendre impossible toute découverte accidentelle de leurs valeurs réelles sur les serveurs. Une variante des signatures algébriques a été alors introduite, les signatures cumulatives. Notre encodage possède notamment des propriétés accélérant diverses recherches de chaînes de caractères, par rapport à celles explorant les mêmes données sans encodage. Certaines recherches se révèlent expérimentalement plus rapides que par des algorithmes fondamentaux bien connus, tels que celui de Karp-Rabin. Nous avons présenté différents algorithmes correspondants aux recherches possibles dans SDDS-2005 tels que la recherche par le préfixe, la recherche partielle d'une chaîne de caractère, la recherche du plus grand préfixe ou encore de la plus grande chaîne commune. Nous avons aussi présenté des méthodes alternatives pour la recherche tell que la recherche par la méthode *XOR* ou par la méthode *n*-Gramme. Cette dernière tire profit des propriétés de signatures cumulatives et s'avèrent très efficace, du moins théoriquement, pour la recherche de longues chaîne.

Le sujet de recherche de cette thèse est au confluent de plusieurs domaines de recherche, notamment la conception de structures de données scalables et distribuées, la recherche de chaînes de caractères, la corruption et l'encodage de données. Dans ce qui suit, nous faisant un bilan général de nos travaux en évoquant les perspectives pour les travaux futurs. Le système résultant devrait aussi assurer différentes qualités tels qu'extensibilité, efficacité, modèle de communication et couplage avec d'autres systèmes

Finalement, notre objectif est atteint. Nos résultats prouvent la scalabilité de notre système. Aujourd'hui, le prototype SDDS-2005 constitue l'une des plus importantes implantations de Gestionnaire réparti de Structure de Données Distribuées et Scalable pour des fichiers SDDS. Ces travaux ouvrent diverses perspectives de recherche que nous détaillons dans les sections suivantes. Enfin, on devrait étudier des applications de nos résultats et techniques, notamment aux grandes bases de données et à l'accélération de serveurs multimédia.

9.1 Bilan Général

Nous avons commencé par l'étude des systèmes de gestion de fichiers distribués classiques en précisant leurs architectures, avantages et inconvénients. Nous avons abordé les méthodes statiques et dynamiques. Certaines de ces dernières présentent plusieurs avantages notamment en ce qui concerne la taille illimitée des fichiers en plus de l'extensibilité des performances d'accès. Les méthodes dynamiques, auxquelles appartiennent les SDDSs, répondent aux inconvénients des systèmes statiques

et apportent plusieurs améliorations, dont une gestion décentralisée des fichiers qui résout le problème du goulot d'étranglement présent dans la plupart des systèmes de gestion de fichiers distribués traditionnels.

Nous avons étendu l'architecture du premier prototype SDDS-2000 pour l'implantation de notre gestionnaire de SDDSs, reposant sur une version multi fichier de RP* et en prenant en compte différentes exigences. Celles ci concernent le traitements concurrentiel de données, la gestion de la mémoire, des accès aux disques ainsi que les méthodes de communication inter-processus disponibles sous Windows 2000/NT, notamment la communication de type UDP, multicast et LPC (Local Procedure Call). Sur ces bases, nous avons abouti au prototype, appelé SDDS-2005, répondant aux besoins exprimés.

Comme pour les fonctions déjà existantes, l'implémentation de nouvelles primitives s'appuie sur le modèle multithread, largement utilisé aujourd'hui, et permettant de décomposer une application en plusieurs modules élémentaires appelés threads. Pour la communication assurant l'interaction entre ces composantes, les protocoles UDP et TCP/IP sont utilisés suivant la taille de ces messages. Ils sont adoptés, en général, pour le transport de messages dans les systèmes répartis et permettent d'assurer l'interopérabilité entre une grande variété de plates-formes. A ce niveau, le modèle des sockets Windows reste le plus performant pour ces transferts.

Nous avons implanté un nouveau schéma de sauvegarde / restauration pour les fichiers SDDS. Une sauvegarde sur disque permet de libérer la mémoire occupée en sauvegardant un fichier peu utilisable. La fonction inverse permet la récupération de données après qu'une panne soit intervenue sur des serveurs de données. L'utilisation de nouvelle forme de signatures, dites signatures algébriques, permet de ne sauvegarder (charger) que les données ayant été modifiées depuis la dernière mise à jour.

Ces signatures sont aussi utilisées dans le schéma de mise à jour. On a étudié deux formes de mises à jour : normales et aveugles. Elles permettent de ne mettre à jour que les enregistrements réellement modifiées. L'utilisation de ces signatures dans notre schéma permet un accès concurrentiel (sans verrouillage) de données par ces clients sans attente ni interblocage.

Nous avons aussi introduit une variante de ces signatures, les signatures cumulatives. Elles permettent la protection de données contre les vues accidentelles du contenu de leurs enregistrements au niveau des serveurs de données. Cela est possible grâce à l'encodage de la partie non clé des enregistrement au niveau du client. Les signatures cumulatives permettent également l'accélération de la recherche par le contenu au sein de nos enregistrements sans décodage local. Dans ce contexte, on trouve plusieurs

algorithmes dans le domaine de la recherche de chaînes de caractères dont les plus importants furent notamment ceux de Karp Rabin, Boyer-Moore ou encore l'algorithme de Knuth-Mooris-Pratt. Nous avons étudié celui de Karp-Rabin comme exemple de comparaisons à notre schéma. Nous avons cité les différents types de recherche possibles tels que la recherche par le préfixe, d'une chaîne de caractères, du plus grand préfixe ou encore de la plus grande chaîne commune. Nous avons aussi présenté une nouvelle technique de recherche, dite recherche par la méthode *n*-Gramme. Cette dernière est basée sur les signatures cumulatives. Elle permet des sauts plus importants. La recherche par le préfixe possède une complexité d'une seule opération. Notre approche est alors parmi les techniques les plus rapides dans le domaine. Par ailleurs, pour certains de ces types de recherche, il n'est pas nécessaire d'envoyer la chaîne recherchée. Seule sa signature sera envoyée aux serveurs de données ce qui constitue un autre avantage pour la sécurité de transmission de données.

Afin de valider notre architecture ainsi que les choix techniques adoptées pour l'implantation de notre prototype, nous avons réalisé plusieurs mesures expérimentales de performance. Les temps obtenus sont stables et proches de la scalabilité idéale. Pour le calcul de signatures algébriques, le temps moyen obtenu est de 0.20 *ms* /MB. Il constitue moins de 10% du temps nécessaire pour le sauvegarde d'un Une mise à jour de 5% de données au niveau d'un fichier maintiens ce gain de stockage à environ 90%. Nous avons comparé nos signatures avec les signatures cryptographiques SHA-1. Les mesures montrent des temps très intéressant. En effet, nos signatures sur 2 Octets s'avèrent suffisant et sont aussi efficaces que les signatures SHA-1. Ces dernières nécessitant 20 Octets d'ou l'intérêt majeurs de nos signatures. La mise à jour utilisant les signatures algébriques est également bénéfique. Une mise à jour normale sur un enregistrement avec une même valeur ne coûtera que 0.28 *ms*/ KO. Le gain est alors de 50% au minimum.

Le temps nécessaire pour l'encodage d'un fichier est d'environs 0.3 *ms* /MB ce qui revient au calcul des signatures cumulatives. La différence n'est pas énorme (14% du temps d'insertion) d'autant plus que le système offre une protection contre les vues accidentelles du contenu sur les serveurs et la protection contre la corruption des données. On a aussi constaté un important gain pour les scans (recherche par le contenu). On retrouve un gain de 30% au minimum par rapport aux signatures algébriques et 20% par rapport à d'autres algorithmes connus tels que Karp-Rabin. Ceci est valable surtout lorsqu'il s'agit de chaînes de grande taille. Les résultats préliminaires utilisant l'algorithme de recherche par la méthode *n*-Gramme ont confirmé les résultats théoriques. Celui ci peut être plus rapide que la recherche par l'algorithme bien connu de Boyer-Moore surtout pour la recherche de longues chaînes.

Tous ces temps sont de l'ordre de plusieurs dizaines de fois inférieure au temps d'accès au disque. Ces résultats confirment les prévisions théoriques. Le temps de stockage, chargement de fichiers, le calcul des signatures ainsi que la recherche par le contenu présentent une scalabilité presque idéale

Enfin, le prototype SDDS-2005 est disponible en téléchargement sur le site CERIA. Notre système est ouvert dans le sens où il fournit une interface sous forme des fonctions externes qui peuvent être utilisées par les applications des utilisateurs pour créer des fichiers et d'y accéder. Les Annexes A et B donnent des indications importantes pour l'installation et l'utilisation de ce prototype.

9.2 Limitations des expérimentations

Les expérimentations réalisées à l'aide de notre prototype présentent les limitations suivantes :

- L'expérimentation a été conduite avec seulement cinq serveurs et deux clients au maximum. Il serait alors intéressant de mener d'autres expérimentations afin d'évaluer les performances avec un nombre de clients et de serveurs beaucoup plus important, d'évaluer l'impact de ces facteurs sur les performances globales du système et d'étudier le comportement de notre gestionnaire sur des multi ordinateurs constitués de centaines, voire de milliers de stations de travail.
- Nous avons supposé une génération des requêtes par les clients suivant une permutation aléatoire prédéfinie. Dans une application réelle, la génération de requêtes peut se faire à des taux différents et peut suivre des distributions différentes. Par conséquent, il est important d'expérimenter différents taux de génération de requêtes et d'autres formes de distributions afin d'évaluer l'impact de ce facteur sur les performances du système.
- Les requêtes utilisées sont simples et consistent seulement à une insertion ou à une recherche de données de taille fixe. Dans une application réelle, les requêtes peuvent être plus complexes en ayant des paramètres et des valeurs de retour de tailles différentes.
- La bande passante disponible entre le client et le serveur n'est pas prise en considération par les stratégies de migration. Dans un système à grande échelle, les stratégies de migration doivent tenir compte de cet aspect afin d'éviter de choisir des serveurs dont les liens réseaux sont congestionnés. Pour cela, il faut disposer d'outils permettant d'évaluer la bande passante disponible à un instant donné entre clients et serveurs.

9.3 Perspectives

Ce travail ouvre de nouvelles possibilités pour de nombreuses applications. Plusieurs problèmes au niveau des systèmes distribués méritent une étude approfondie. Parmi ces perspectives, nous présentons quelques unes dans ce qui suit.

9.3.1 Accélération de Calcul de Signatures Algébriques

On peut envisager l'utilisation de plusieurs variantes pour l'accélération du calcul de signatures algébriques. D'abord, l'utilisation de caches L1 et L2 du Pentium pour mettre le contenu de l'une ou des deux tables *Log* et *Antilog* décrites au chapitre 4. Alternativement, si possible aussi y cacher la page courante. Cette approche est possible par l'utilisation de l'instruction assembleur '*prefetch*' sous le compilateur C de Microsoft. Ensuite, Le calcul d'une signature peut être modifié pour être aisément factorisé en schéma populaire de Horner. Le calcul de signatures inversées offre un encodage plus rapide de données. Ces signatures seront fort utiles notamment dans le cas ou la vitesse d'encodage prime sur celle de la recherche de chaînes

9.3.2 Protection contre la corruption dans SDDS-2005

L'utilisation de signatures algébriques permet la détection d'une corruption dans un enregistrement et de déterminer avec précision la partie dans laquelle s'est produite cette corruption [LMS05d]. Celle-ci est normalement non observée au niveau des clients et des serveurs de données. Ce disfonctionnement est d'autant préoccupant dans la mesure où les autres enregistrements appartenant à d'autres cases, seront à leurs tours affectés par ce disfonctionnement. Ceci est valable pour la plupart des schémas utilisant le calcul de parité.

Pour remédier à des anomalies provenant des parties non clé, nous pouvons développer une approche simple utilisant les signatures algébriques dans GF (2^{16}). Le client insert des données additionnelles (méta- données) accompagnant cet enregistrement. Il s'agit de la signature de l'enregistrement. Pour chaque enregistrement, le client procède à l'insertion de checksums représentant la signature algébrique sur n caractères pour chaque 255 Octets. Pour l'encodage partiel par les signatures cumulatives, ce n équivaut à 1 (suffisant en pratique). Le client calcule ses 'checksums' durant l'encodage des enregistrements en procédant par des 'XOR' successifs des caractères encodés dans GF (2^8). Ce checksum est simple. Il équivaut à $p"_{255}$. Ce calcul n'est pas fait de la même manière lorsque ces 'checksums' sont calculés à partir des signatures algébriques cumulatives. La corruption dans ce cas est plus probable et une erreur dans $p"_i$ conduira obligatoirement à d'autres erreurs au niveau de $p"_{i+1}$. Une

solution consiste à calculer le 'checksum' du 2ème caractère de la signature S_1S_2 par partie de 255 Octets, avec $S_1 = \beta^n$ pour une partie de n Octets. On ne calculera alors que la signature S_2 qu'on sauvegardera au niveau du client.

L'utilisation d'une signature à n caractères permet la détection de corruption survenue au niveau de n caractères à coups sur vu que cette probabilité devient très petite. La probabilité de non détection devient alors $1/2^{nf}$. L'approche utilisant les signatures cumulatives étant plus complexe mais aussi plus efficace. A la réception d'enregistrements provenant des serveurs suite à la satisfaction d'une requête quelconque, le client recalcule ces checksums durant le processus de décodage. Ce calcul concerne le contenu de ces enregistrements reçus. Ce processus n'est pas coûteux pour la méthode utilisant les signatures algébriques mais un recalcul de S_2 est nécessaire dans le cas d'utilisation de signatures algébriques cumulatives. S'il y'a égalité avec les checksums sauvegardés par le client à l'encodage, l'enregistrement est déclaré 'sans corruption' avec la probabilité d'erreurs évoquée plus haut. Dans le cas contraire (pas d'égalité), cela signifie qu'il y'a eu corruption sur les données. Le client avertit le serveur de données concerné en identifiant la partie exacte contenant une corruption dans le but de corriger celle ci.

Finalement, la protection contre la corruption n'est pas très coûteuse pour l'encodage partiel avec une probabilité de non détection de 1/255 seulement. La technique utilisant les signatures cumulatives nécessitera un couple de 'XOR' et quelques opérations en plus. Cela rend le processus plus complexe mais plus rapide. L'utilisation d'un deuxième caractère dans la signature cumulative dans GF (2^8) ou GF (2^{16}) est surtout plus efficace du moment que cette probabilité devient 1/64K pour n=2.A chaque rajout de ce 'checksum', l'enregistrement est élargi de moins de 0.5%. Ce qui n'est pas coûteux du point de vue 'architecture'.

9.3.3 Compression Delta des Mises à Jour

Nous pouvons discuter l'utilisation de ces signatures pour la compression différentielle des mises à jour distribuées de données. Cette méthode permet aussi la détection de l'endroit exact de la mise à jour évitant le renvoi aux serveurs, des données n'ayant pas subi de modifications. Elle permet également la protection contre la corruption de données.

Cette technique permet la détection de modifications survenues dans un enregistrement à la suite d'une mise à jour de celui ci. Afin de pouvoir détecter cette mise à jour, notre approche est basée sur les signatures algébriques. Cela est possible grâce à l'ajout, pour chaque enregistrement, de 'références' ou 'méta données' dans son '*image avant*'. On appellera le contenu de cet enregistrement avant sa mise à

jour 'l'image avant'. Son contenu après la mise jour est appelée '*image après*'. Ceci est d'autant bénéfique que ces signatures sont plus petites que les données mises à jour et souvent, beaucoup plus petite.
La technique de compression résulte du processus entraînant l'envoi de l'enregistrement et les signatures cumulatives rajoutée à ce dernier. Durant la compression différentielle [LMS05d, A&all02], le traitement sur l'enregistrement se fait au niveau du client. Celui ci traite l'image après provenant de l'application tout en ayant l'image avant provenant du serveur de données suite à une opération de recherche. Le serveur analyse (décompresse) chaque enregistrement obtenant les différentes signatures correspondantes à chaque image avant. Un algorithme de détection de mise à jour est appliqué. Il consiste en la comparaison de ces signatures. D'abord, il procède à l'extraction de la chaîne de caractères reçue au sein de l'image après puis il met à jour l'enregistrement en question.

La compression différentielle est efficace surtout lorsque la mise à jour concerne une petite fraction de l'enregistrement. Ceci est souvent le cas dans les bases de données. Cette compression est également utile pour la sécurité des données. En rajoutant ces signatures aux enregistrements, il est pratiquement Impossible de déterminer le contenu de ceux ci s'ils sont interceptés sur le réseau. De plus, la compression différentielle de données peut être utile pour notre module de stockage dans les SDDS ou encore dans les applications 'Client Cache Management'. Le client met à jour son 'cache' et envoi uniquement la partie mise à jour en se référant à la signature reçue du serveur. Celui ci envoi à son tour la partie changée seulement. Dans SDDS-2005, Nous aborderons la compression différentielle non seulement lors de la mise à jour mais aussi lors de l'encodage d'enregistrements à travers l'utilisation des signatures algébriques cumulatives. Les propriétés de ces signatures font que cette compression différentielle est plus rapide que les autres méthodes d'autant que cet encodage de données est utile pour la protection contre les vues accidentelles des données et l'accélération de différents types de recherche de chaînes de caractères. La compression naïve consiste à examiner chaque enregistrement de façon séquentielle. Pour une chaîne de caractères à compresser de taille l. Nous pouvons étudier la compression par préfixe, suffixe et 'combiné' (appelée aussi overall combinant la compression par le préfixe et le suffixe) [LMS05d]. La compression par préfixe permet la détection du préfixe commun le plus long entre l'image avant et l'image après de l'enregistrement. Celle par suffixe fait de même par rapport au suffixe. La compression 'combiné' cherche à remplacer chaque chaîne commune plus longue qu'une certaine chaîne de référence. Elle combine les deux techniques. Une comparaison pourra être faite aux travaux les plus récents qui figurent dans [A&all02]. On retrouve notamment le schéma appelé ABFLS et qui s'appuie sur l'algorithme de Karp-Rabin.

9.3.4 Etude plus Approfondie pour la Recherche

Il s'agit d'une part de l'étude théorique et expérimentale plus approfondie des algorithmes de recherche présentés dans le chapitre 7 telle que la déterminaison de la complexité de recherche pour un fichier réel. D'autre part, de comparer ces algorithmes avec d'autres algorithmes présentés dans [CL00] notamment ceux de Boyer-Moore [BM77] ou de Knuth-Morris-Pratt [BMP77]. On pourra aussi comparer notre algorithme à d'autres versions de l'algorithme Karp-Rabin telle que celle de Crochemore [CL03]. Diverses techniques de résolution de collisions (discutées au chapitre 7) peuvent être aussi implémentés

9.3.5 Gestion de Transaction

Les signatures algébriques peuvent être exploités pour la gestion de transaction. Une approche est présentée dans [SH04]. D'autres travaux pourront consister à l'application des SDDS aux grandes bases de données multimédia, réseaux de stockage, entrepôts de données, aux caches Web où les volumes de données sont très importants ou encore à l'accélération de serveurs multimédia. Dans ces domaines, des extensions aux noyaux de notre système devraient être introduites pour permettre une gestion efficace des transactions.

9.3.6 Application de notre prototype

Notre prototype a été annoncé sur DbWorld. Nous avons été contacté par quelques usagers qu'on a bien voulu aider. On peut envisager de nombreuses applications dans le cadre P2P ou Grid. Une application intéressante apparaît. Elle consiste à l'exploration de bases DNA expérimentées dans la fin du chapitre 9. Une autre application particulièrement intéressante, celle du projet e-Gov. Dans ce cadre, nous allons étudier la gestion de documents électroniques stockés dans SDDS-2005 autant qu'un élément du répositoire virtuel. Nous nous intéressons notamment à la sauvegarde de documents XML administratifs accessible fréquemment. Cette application fait partie du projet e-Gov de la CEE dans lequel nous participons [LMS06].

9.4 Conclusion générale

Nos résultats prouvent la scalabilité de notre système. Aujourd'hui, le prototype SDDS-2005 constitue l'une des plus importantes implantations de Gestionnaire réparti de Structure de Données Distribuées et Scalable pour des fichiers SDDS. Ces travaux ouvrent diverses perspectives de recherche et de nouvelles possibilités pour de nombreuses applications. Nous avons présenté quelques unes.

Enfin, ce manuscrit résume notre travail de quatre ans qui nous a enrichis dans différents domaines touchés par cette thèse. Nous espérons qu'il sera utile à la communauté scientifique.

Bibliographie

[A01] Aberer, K.. P-Grid: Self-organizing Access Structure for P2P Information Systems. COOPIS 2001, Trento, Italy.

[A&al02] Ajtai, M., Burns, R., Fagin, R, Long, D., Stockmeyer; L. Compactly Encoding Unstructured Inputs with Differential Compression. Journal of the ACM, Vol. 49, No. 3, May 2002, pp. 318–367.

[A&al99] Atzeni, P., Ceri, Stefano, Paraboschi, S. and Torlone, R. Database Systems: Concepts, Language and Architectures. McGraw-Hill, 1999

[ACP95] Anderson, T. ., Culler, D. E., and Patterson, D. A Case for NOW (Network of Workstations). IEEE Micro, 12(1):54-64, February 1995.

[AD01] Aberer, K., Despotovic, Z. Managing Trust in a Peer-2-Peer Information System. Proceedings of the Ninth International Conference on Information and Knowledge Management (CIKM 2001) 2001.

[ADW99] Ailamaki, A. & al. DBMSs on a modern processor: Where does time go. In Proceedings of the 24th VLDB Conference, 1999.

[AH00] Adar, E. and Huberman, B. A. Free riding on Gnutella Techinical report, Xerox PARC, 10 Aug. 2000.

[B00] Benga, N. E. Implémentation d'un client RP* en java. Université de Dakar, avril 2000.

[B00a] F. Bennour, Un Gestionnaire de Structures Distribuées et Scalables pour les multi-ordinateurs Windows : fragmentation par hachage, Rapport de thèse, Université Paris Dauphine

[B92] Bloomer, John. Power Programming with RCP. O'Reilly & Associates, 1992.

[B93] Broder A Z. Some applications of Rabin's fingerprinting method. In Capocelli, De Santis, and Vaccaro, (ed.), Sequences II: Methods in Communications, Security, and Computer Science, pages 143--152. Springer-Verlag, 1993.

[B96] Fibre Channel: Gigabit Communication and I/O for Computer Networks. McGraw Hill, New York, 1996.

[B&al00] W. J. Bolosky, S. Corbin, D. Goebel, J. R. Douceur; Proceedings of the 4th USENIX Windows Systems Symposium, 2000, pp. 13-24.

[B&al01] International Conference on Measurement and modeling of computer systems, 2000, pp.33-43.Bolosky, W. J., Douceur, J. R., Ely, D., Theimer, M. Proceedings of the

[BDE00] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. SIGMETRICS 2000.

[B&al00] Bennour, F. Diène, A. W. Ndiaye Y. Litwin, W. Scalable and Distributed Linear Hashing LH^*_{LH} under Windows NT. SCI-2000 Orlando, Florida, USA. July 23-26, 2000.

[BKV98] L. Bouganim, O. Kapiskaia, P. Valduriez : Dynamic Memory Allocation for Large Query Execution. Network and Information Systems Journal, Hermès, 1(6), 1998

[B&al93] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Research Report 111, Systems Research Center, Digital Equipment Corporation, September 1993.

[BM72] Bayer, R. and McCreight, E. Organization and Maintenance of large ordered indexs. Acta Informatica, 1:173-189, 1972.

[BM77] Boyer R.S and Moore J S. A fast string-searching algorithm. ACM 1977.

[C00] B. Callaghan, NFS Illustrated. Addison Wesley Publishing Co., Reading, Mass., 2000.

[C05] Site du CERIA http://ceria.dauphine.fr/

[C79] Comer, D. The Ubiquitous B-tree. Computing Surveys, 11(2):121-137, 1979.

[C88] P. Y. Chevalier. Persistance et Disponibilité dans les Systèmes Répartis : Application à Guide. Thèse de Doctorat, Université Joseph Fourrier, Grenoble I, novembre 1988.

[C97] M. Crochemore: Off-line serial exact string searching, in Pattern Matching Algorithms, A. Apostolico and Z. Galil, eds., Oxford University Press, New York, 1997) Chapter 1, pp 1–53.

[CDK94] G. Colouris, J. Dollimore, T. Kindberg, Distributed Systems - Concepts and Design, pp 233-243, Addison-Wesley, Second Edition (1994).

[CK88] T.L. Casavant and J.G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. IEEE Transactions on Software Engineering, 14(2), pp. 141-154, Februrary 1988.

[CL00] http://www-igm.univ-mlv.fr/~lecroq.htm. Exact String Matching Algorithm. Christian Charras, Thierry Lecroq.

[CL03] M. Crochemore and T. Lecroq, Pattern Matching and Text Compression Algorithms, in The Computer Science and Engineering Handbook, A.B. Tucker, Jr, ed., CRC Press, Boca Raton, 2003.

[CL03a] Crochemore M, Lecroq.T. Pattern matching and text compression algorithms. Institute Gaspard Monge.

[CR94] Crochemore, M., Rytter, W., 1994, Text Algorithms, Oxford University Press

[CS01] Comer, D. E., Stevens, D. L. Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications, Linux/Posix Sockets Version 1/e. Published September 2000 by Prentice Hall.

[CTO00] Condor Team Overview of the Condor High Throughput Computing System, 2000. http://www.cs.wisc.edu/condor/overview/

[D01] Contribution à la Gestion de structures de Données Distribuées et Scalables. Thèse de doctorat. Aly Wane Diène. Univ Paris Dauphine. 2001.

[D84] DeWitt, D. J. and al. Implementation Techniques for Main Memory Database System. Proc ACM SIGMOD Conf, Boston, MA, June 1984.

[D93] Devin, D. Design and implementation of DDH : A distributed dynamic hashing algorithm. In Proc. of the 4th Intl. Foundation of Data Organization and Algorithms (FODO), 1993.

[D97] Diène, A. W. Organisation interne des SDDS RP*. Mémoire de DEA. Université de Dakar, 1997.

[D99] Diène, A. W. Contrôle du flux sous SDDS-2000. Rapport de recherche, CERIA Paris 9 Dauphine, 1999.

[DL01] Diène, A. W. Litwin, W. Performance Measurements of RP*: A Scalable Distributed Data Structure for Range Partitioning. 2000 Intl. Conf. on Information Society in the 21st Century: Emerging Techn. and New Challenges. Aizu City, Japan, 2000.

[DL01] Diène, A. W. Litwin, W. Implementation and performance measurements of the RP* Scalable Distributed Data Structure for Windows Multicomputers. PADDA 2001. Munich, Germany, April 19th-20th, 2001.

[DNB00a] Diène, A. W. Ndiaye, Y., Bennour, F. Une architecture pour une gestionnaire de structures de données scalable et distribuées RP* sous Windows NT. Rapport de recherche, CERIA Paris 9 Dauphine, 2000.

[DNB00b] Diène, A. W. Ndiaye, Y., Bennour, F. Couplage de Structures de Données Distribuées et Scalables avec un SGBD relationnel-Objets. Rapport de recherche, CERIA Paris 9 Dauphine, 2000.

[E87] Eich, M. H. A Classification and Comparison of Main Memory Database Recovery Techniques. Proc 3rd Int Conf on Data Engineering, Los Angeles, CA, February 1987.

[E96] Yves Eichenlaub. Problèmes effectifs de théorie de Galois en degré 8 à 11. Thèse. Université de Bordeaux 1996.

[F02] Farsite : http://research.microsoft.com/research/sn/Farsite/

[F01] Foster, I. The Anatomy of the Grid: Enabling Scalable Virtual Organizations, IJSA, 2001.

[FRS93] Fahl, G., Risch, T., Sköld, M. AMOS - An Architecture for Active Mediators. Workshop on Next Generation Information Technologies and Systems (NGITS'93), Haifa, Israel, June 1993. http://www.dis.uu.se/~udbl/amos/doc/amos_concepts.html

[G01] Gardarin G. Bases de Données. Objet et Relationnel. Eyrolles. Troisième tirage 2001.

[G88] Gray, J. The Cost of Messages. 7th ACM Symp. on Principles of Distributed Systems, 1988.

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

[G99] Gray, J. Turing Award Lecture: What Next? ACM Federated Research Computer Conference, Atlanta, Georgia, 4 May 1999.

[GM00] G. A. Gibson, R. V. Meter, Network Attached Storage Architecture. Communication of ACM, November 2000/Vol.43, No.11.

[Gn01] Gnutella : http://gnutella.wego.com

[GS92] Garcia-Molina, H. and Salem, K. Main Memory Database Systems: An Overview. IEEE Trans on Knowledge and Data Engineering, vol 4, no 6, Decembre 1992.

[GS97] Garcia-Molina, H. and Salem, K. High Performance Transaction Processing with Memory Resident Data. Proc in Workshop on High Performance Transaction System, Paris, December 1987.

[H03] Hamadi, B. Suppressions et Mises à Jour dans le Système SDDS-2000. CERIA Res. Report, 2003-04-4.

[H&al88] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. ACM Transaction on Computer System, 6(1):51-81, February 1988.

[I04] Ingold Rolf.2004 Programmation 3A: Algorithmique. Recherche de sous-chaînes. Université of Fribourg.2004

[IC96] Inktomi Corporation. The Inktomi Technology Behind HotBot, May 1996. http://www.inktomi.com/products/network/traffic/tech/clustered.html

[j&al06] Jagadish H, Chin Ooi B, Hieu Vu Q, Rong Z, Zhou A. VBI-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes. ICDE'06. 22nd international Conference on Data Engineering. Atlanta. Georgia. USA. 2006.

[J88] Jacobson, V. Congestion Avoidance and Control, Computer Communication Review,

vol. 18, no 4, pp. 314-329 (Aug.), 1988.

[J94] Jagadish, H. V., and al. Dali: A High Performance Main Memory Storage Manager. Proc VLDB Conf, Santiago, Chile, Septembre 1994.

[K73] Knuth, D. E. The Art of Computer Programming: Sorting and Searching, volume 3. Addison-Wesley Publishing Company, Reading, Massachussetts, 1973.

[K95] Kim, W.(ed.), Modern Database Systems: the Object Model, Interoperability and Beyond, ACM Press and Addison-Wesley, New York, 1995.

[K&all] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. OceanStore: An Architecture for Global-Scale Persistent Storage. Appears in Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000.

[KLR94] J. S. Karlson, W. Litwin and T. Risch. LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers. In Advances in Database Technology - EDBT'96, pages 573-591, Avignon, France, March 1996. Springer.

[K98] Jonas S. Karlsson. hQT*: A Scalable Distributed Data Structure for High-Performance Spatial Accesses. Url: citeseer.ist.psu.edu/karlsson98hqt.html

[KMP77] KNUTH, D.E., MORRIS (Jr) J.H., PRATT, V.R., 1977, Fast pattern matching in strings, SIAM Journal on Computing 6(1):323-350

[KR87] Karp,R.M.,Rabin,M.O.1987. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development, Vol. 31, No. 2, March 1987: 249–260.

[KS86] Korth, H. F., Silberchatz, A. Database System Concepts. Mc Graw Hill Inc., New York, 1986.

[KTM84] M. Kitsuregawa, H. Tanaka and T. Moto-Oka. Architecture and performance of relational algebra machine GRACE. In Proc. of the Intl. On Prallel Processing, Chicago, 1984.

[KW94] B. Kroll and P. Widmayer. Distributing a Search Tree Among a Growing Number of Processors. In ACM-SIGMOD International Conference On Management of Data, 1994.

[L&al06] Litwin W, Mokadem R, Rigaux P, Schwarz T 'Pattern Matching Using Cumulative Algebraic Signatures and n-Gramme Sampling'.To appear. 3rd International Workshop on Grid and Peer-to-Peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems W05 - GLOBE '06. Krakow, Poland 4 - 8 September 2006

[L&al06a] Litwin W, Mokadem R, Rigaux P, Schwarz T 'Pattern Matching Using Cumulative Algebraic Signatures and n-Gramme Sampling'. Ceria Research Report 2006-3-26

[L00] M. Ljungstrm, Implementing LH*RS: a Scalable Distributed Available Data Structure, Master Thesis, CS Dep., U. Linkoping, Feb. 2000.

[L78] P. A. Larson. Dynamic Hashing. BIT, pages 184-210, 1978.

[L80] Litwin, W. Linear Hashing : a new tool for file and tables addressing. Reprint from VLDB-80.

[L92] Lecroq, T., 1992, Recherches de mot, Ph. D. Thesis, University of Orléans, France

[L95] Litwin, W. Redundant Arrays of LH* files for high availability and security. Technn Note GERM & Distributed Inf. Techn. Dep HPL Palo Alto, Sept. 1995.

[LC00] Handbook of exact String Matching Algorithms. Thierry Lecroq, Christian Charras.2000

[LC85] Lehman, T., Carey, M. A Study of Index Structures for Main Memory Database Management Systems, UW CS Tech Rep # 605, July 1985.

[LC86] Lehman, T. J., and Carey, M. J. A Study of Index Structures for Main Memory Database Management Systems. Proc VLDB Conf, Kyoto, August 1986.

[LKO00] Lee, L. L., Kitsuregawa, Ooi, B. C. Towards Self-Tuning Data Placement in Parallel Database Systems. ACM SIGMOD, 2000.

[LKR96] Litwin, W., Karlsson et Risch J. LH*lh: A Scalable High Performance Data Structure for Switched Multicomputers. T. Intl. Conf. on Extending Database Technology, EDBT-96, Avignon, March 1996.

[LLM88] M.J. Litzkow, M. Livny, and M. W. Mutka. Condor : a hunter of idle workstation. In Proc. Of the 8th International Conference on Distributed Computing System, IEEE, san Jose, California, pp.104-111, June 1988.

[LM92] M.J. Litzkow, M. Solomon. Supporting checkpointing and process migration outside the UNIX kernel. Inn USENIX Winter Conference, San Francisco, California, pp. 283-290, Janurary 1992.

[LMRS99] Litwin, W., Menon, J., Risch, T., Schwarz, Th. Design Issues For Scalable Availability LH* Schemes with Record Grouping. DIMACS Workshop on Distributed Data and Structures, Princeton U. Carleton Scientific, (publ.) 1999.

[LMS05a] W Litwin, R.Mokadem & Th.Schwarz. Pre computed Algebraic Signatures for faster string search. Protection against incidental viewing and corruption of Data in an SDDS. Proceeding of third international workshop and co-located

with the 31st international conference on Very Large Data Bases (DBISP2P 2005). Trondheim, Norway. August 2005.

[LMS05b] W Litwin, R.Mokadem & Th.Schwarz. Cumulative Algebraic Signatures for fast string search. Protection against incidental viewing and corruption of Data in an SDDS. Springer in LNCS series. Proceeding DBISP2P 2005)

[LMS05c] W Litwin, R.Mokadem & Th.Schwarz Pre-computed Algebraic Signatures for faster string search. Protection against incidental viewing and corruption of Data in an SDDS. CERIA Research. Report. 2004-10-23

[LMS05d] Differential Compression Using Cumulative Algebraic Signatures in an SDDS W. Litwin, R. Mokadem & Th. Schwarz. CERIA Research. Report. 2004-10-23

[LMS04] Litwin W, Moussa. R., Schwarz T., LH*RS: A Highly Available Distributed Data Storage System. Intl. Conf. On Management of Very Large Databases, VLDB-04, Toronto 2004.

[LMS03a] W Litwin, R. Moussa, T Schwarz $LH*_{RS}$ – A Highly-Available Scalable Distributed Data Structure. ACM-TODS, Sept. 2005.

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

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

[LN96] Litwin, W., Neimat, M-A. High-Availability LH* Schemes with Mirroring. Intl. Conf. on Coope. Inf. Syst. COOPIS-96, Brussels, 1996.

[LNS93a] Litwin, W. Neimat, M-A., Schneider, D. LH* : Linear Hashing for Distributed Files. ACM-SIGMOD Int. Conf. On Management of Data, 1993.

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

[LNS94] Litwin, W. Neimat, M-A, Schneider, D. RP*: A Family of Order- Preserved Scalable Distributed Data Structures. 20th Intl. Conf. On Very Large Data Bases (VLDB), 1994.

[LS00] Litwin, W. Schwartz, Th. LH*RS: A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes (CERIA Res. Rep. & ACM SIGMOD 2000, Dallas)

[LS04] Litwin, W., Schwarz, Th. Algebraic Signatures for Scalable Distributed Data Structures. IEE Intl. Conf. On Data Eng., ICDE-04.

[LSS02] Litwin, W., Scheuermann, P., Schwarz Th. Evicting SDDS-2000 Buckets in RAM to the Disk. CERIA Res. Rep. 2002-07-24, U. Paris 9, 2002.

[LS86] Lanin, V., Shasha, D. A Symmetric Concurrent B-tree Algorithm. Proc. FJCC, 380-389. (1986).

[LS90] Levy, E., Silberschatz, examples. A.. Distributed file systems : Concepts and ACM Computing Surveys, 22(4), December 1990.

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

[LSC92] Lehman, T. J., and Shekita, E. J., and Cabrera, L. F. An Evaluation of Starburst's Memory Resident Database Storage Component. IEEE Trans on Knowledge and Data Engineering, vol 4, no 6, December 1992.

[LT96] E. Lee, C. Thekkath. Petal: Distributed virtual disks. 7th ASPLOS, p. 84. 92, Oct 1996.

[LT97] Litwin, W., Risch, T. LH*g : a High-Availability Scalable Distributed Data Structure through Record Grouping. Rees Rep. CERIA, U. Dauphine & U. Linkoping (May. 1997)

[LVZ93] R. Lanzelotte, P. Valduriez, M. Zait: On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces. Int. Conf. on VLDB, Dublin, August 1993

[LY81] Lehman, T., Yao, S. Efficient Locking for Concurrent Operations on B trees. ACM TODS, 6(4) December 1981.

[M02] Mokadem R, Stockage de données en utilisant les signatures algébriques dans les SDDS. Mémoire DEA. Université Paris Dauphine. Septembre 2002

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

[M98] Microsoft. - DCOM: A Technical Overview. - Technical report, Microsoft Corporation, 1998. http://www.microsoft.com/ntserver/appservice/

[ML06] Mokadem R, Litwin W "String-matching and update through Algebraic Signatures in Scalable Distributed Data Structures". 3rd International Workshop on "P2P Data Management, Security and Trust (PDMST'06)" Krakow, Poland 4 - 8 September 2006

[MLS03] R Mokadem, W Litwin, Thomas Schwarz.: "Disk Backup Through Algebraic Signatures in Scalable and Distributed Data Structures", Proceedings of the Fifth Workshop on Distributed Data and Structures, Thessaloniki, June 2003 (WDAS 2003). http://ceria.dauphine.fr/Riad/Article_storage-wdas_wl.pdf

[MS97] MacWilliam, F. J., Sloane, N. J. A. The Theory of Error Correcting Codes, Elsevier/North Holland, Amsterdam, 1997.

[N01] Napster : http://www.Napter.com

[N&al00] Ndiaye, Y. Diène, A. W., Litwin, W. Risch, T. Scalable Distributed Data Structures for High-Performance Databases, WDAS 2000, Italia.

[N&al01] Ndiaye, Y. Diène, A. W. Litwin, W. Risch, T. AMOSSDDS: A Scalable Distributed Data Manager for Windows Multicomputers. 14th International Conference on Parallel and Distributed Computing Systems(PDCS-2001). Richardson, Texas, USA, 8-10/8, 2001.

[NOW96] NOW : Network of Workstations. Building system support for using a network of workstations as a distributed supercomputer on a buildingwide scale. http://http.cs.berkeley.edu/projects/

[O01] OceanStore : http://news.cnet.com/news/0-1003-201-4424563-0.html

[OHE96] Orfali, R., Harkey, D., Edwards, J. Essential Client/Server Survival Guide, 2nd Edition, John Wiley & Sons? Inc., 1996.

[OMG98] OMG. - The Common Object Request Broker: Architecture and Specification - Revision 2.2. - Technical report, OMG Document, 1998.

[ÖV99] Özu M T, Valduriez P. Principles of Distributed Databases Systems. Prentice-Hall, Englewood Cliffs, 666 pages, N.J, 1999.

[PGK88] Patterson, D. A., Ginson. G., Katz, R. A case for Redundant Arrays of Inexpensive Disks, Proc. of ACM SIGMOD Conf, pp. 109-106, June 1988.

[P & al01] P. Pucheral, L. Bouganim, C. Bobineau, P. Valduriez: PicoDBMS : Scaling down database techniques for the Smartcard. The VLDB Journal, special issue on Best Papers from VLDB2000, 10(2-3),2001

[P& al94] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In Proceedings of the Summer USENIX Conference, pages 137-152, June 1994.

[PV98] E. Pacitti, P. Valduriez: Replicated Databases: concepts, architectures and techniques, Network and Information systems Journal, Hermès, 1(3), 1998

[R00] Risch, T. AMOS-II External Interfaces. Dept. of Information Science, Uppsala University, Feb. 2000. http://www.dis.uu.se/~udbl/amos/doc/external.pdf

[R93] Richter, J. Memory-Mapped Files in Windows NT Simplify File, Manipulation and Data Sharing. Microsoft Development Library 1993.

[R97] Richter, J. Advanced Windows. 3d ed. Redmond, Wash.: Microsoft Press, 1997.

[RJK99] Risch, T., Josifovski, V., Katchaounov, T., AMOS II Concepts. Dept. of Information Science, Uppsala University, Nov. 1999.

[RR00] Rao, J., Ross, Kenneth A. Making B+-Trees Cache Conscious in Main Memory. ACM SIGMOD, 2000.

[S&al00] Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, Hari. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. Proceedings of the ACM SIGCOMM, 2001.

[S&al04] T. Schwarz, Q. Xin, E. Miller, D. Long, A. Hospodor, and S. Ng: Disk Scrubbing in Large Archival Storage Systems. 12th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems. (MASCOTS) 2004.

[S&al85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Desing and Implementation of the Sun Network File System. In Proceedings of the UNSENIX 1985 Summer Conference, El Cerrito, CA, USA, June 1985.

[S&al98] Seck, M.T., Ndiaye, S., Diène, A. W. Litwin, W., Levy, Gérard, Implémentation d'une bibliothèque de primitives d'accès à un fichier scalable et distribué RP*. CARI'98, Dakar.

[S01] SETI@home : http://setiathome.ssl.berkeley.edu/

[S75] Steel, G. L. Multiprocessing compactifying garbage collection. ACM Commun. 18, 9, pp 495-508 (CGP), September 1975.

[S85] Sagiv, Y. Concurrent Operations on B-trees with Overtaking. Proc. 4th Symp. On PODS, 28-37 (1985).

[S88] Salzberg, B. J. File Structures: an Analytic Approach. Prentice Hill, Englewood Cliffs NJ, 1988.

[S89] Satyanarayanan, M. & al, CODA: A Highly Available File System for a Distributed Workstation Environment, Proceedings of the Second IEEE Workshop on Workstation Operationg Systems, Sep. 1989, pages 447-459.

[S93] Stevens W. R. TCP/IP Illustrated, Volume 1. Addison Wesley, Reading, Massachusetts, 1993.

[S95] Souleïmane, R. Protocole de Communication pour les Structures de Données Scalables et Distribuées. Rapport de Stage de DEA 127, Université Paris IX Dauphine, sep. 1995.

[S96] Sinha, A. k. Network Programming in WINDOWS NT. Addison-Wesley Publishing Company. 1996.

[S98] Stevens W. R. UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI, Second Edition. Prentice Hall, Englewood Cliffs, NJ, 1998.

[SBB90] T. Schwarz, R. W. Bowdidge, and W. A. Burkhard, "Low cost comparisons of file copies," in Proc. Int. Conf. Distributed Comput. Syst., Paris, France, pp. 196--202, May 1990. http://citeseer.ist.psu.edu/schwarz90low.html

[SDDS] SDDS-bibliography. http://ceria.dauphine.fr/ SDDS-bibliographie.html

[SC83] Shu Lin & Daniel j Costello Jr. Econtrol Coding. Fodamentals ans Applications. Prentice Hall. New jersey. 1983.

[SH95] Secure Hash Standard Processing Standards publication.FIPS PUB 180-1 1995 April

[SH04] Thomas Schwarz, JoAnne Holliday: A Signature Based Concurrency Scheme for Scalable Distributed Data Structures. Workshop on Distributed Data and Structures, WDAS'04, Lausanne, 2004

[SM98] SunMicroSystems. - Enterprise JavaBeans Technology. - Technical report, Sun MicroSystems, 1998. http://java.sun.com/products/ejb/

[SNT04] Suel T, Noel P, Trendafilov D. Improved File Synchronisation Techniques for Maintaining Large Replicated Collections over Slow Networks. ICDE 2004.

[SPW90] C. Severance, S. Pramanik, and P. Wolberg. Distributed Linear Hashing and Parallel Projection in main memory database. In Proc of VLDB, 1990.

[T00] TimesTen Programmer's Guide. TimesTen Performance Software. http://www.timesten.com

[T92] Tanenbaum, A. S. Modern Operating Systems, pp 573-579, Prentice Hall, (1992).

[T95] Tanenbaum, An. S. Distributed Operating Systems. Prentice Hall, 1995.

[T99] The TimesTen Team: In-Memory Data Management fo Consumer Transactions, The Timesten Approach. Proc ACM SIGMOD Conf, Philadelphia PA 1999.

[Ter88] Teradata Corporation. DBC/ 1012 data base computer concepts and facilities. Technical Report Teradata Document C02-001-05, Teradata corporation, 1988.

[TML97] Thekkath C, Mann T, Lee E. Frangipani: A Scalable Distributed File System. In Proceedings of the 16th ACM Symposium on Operating Systems Principes, St.-Malo, France, October 1997.

[VBW98] Vingralek R., Breitbart, Weikum Y, Snowball G: Scalable Storage on Networks of Workstations with Balanced Load. Distributed and Parallel Databases 6(2): 117-156 (1998)

[V93] Valduriez P: Parallel Database Systems: open problems and new issues. Int. Journal on Distributed and Parallel Databases, 1(2), 1993

[V93a] Valduriez P: Parallel Database Systems: the case for shared-something. Keynote address, IEEE Int. Conf. on Data Engineering, Vienna, Austria, April 1993.

[W76] Wadler, P. Analysis of an algorithm for real time garbage collection. ACM Commun. 19, 9, pp 491-500 (AP), September 1979.

[W92] Wilson, Paul R. Uniprocessor garbage collection techniques. In Proc int. Workshop on Memory Management, number 637 in Lectures Notes in Computer Sciences, Saint-Malo (France), September 1992. Springer-Verlag.

[WBW94] R. Wingralek, Y. Breitbart, and G. Weikum. Distributed file organization with scalable cost/performance. In Proc of ACM-SIGMOD, May 1994.

[Y& all] Yagoub K, Florescu D, Issarny V, Valduriez P : Caching Strategies for Data-intensive Web Sites. Int. Conf. on VLDB, Cairo, 2000.

[Z04] Zeggour D. Scalable Distributed Compact Trie Hashing" (CTH*) National Computing High School, Algiers. Information & Software technology. Dblp. Volume 46.923-935

SDDS-2005 V 1.0 Interface Functions

Riad Mokadem

Riad.Mokadem@Dauphine.fr

CERIA

University Paris 9 Dauphine

Place du Mal. De Lattre de Tassigny, 75775 Paris Cedex 16, France

SDDS-2005 V 1.1 is new release of our 1st prototype manager of Scalable Distributed Data Structures (SDDSs). It runs on local networks of Windows 2000 or Windows XP machines. The application manipulates data through an **SDDS-2005 client** on its computer. Data reside at the **SDDS-2005 servers**. Servers keep data in the distributed RAM, for access times up to **hundred times faster than to disk files**. The data distribution is scalable. Your file may spread at any number of SDDS-2000 servers you create at your network. The scaling is transparent to the application. *These capabilities are unique to SDDS-2005 at present.*

SDDS-2005 V 1.1 provides the scalable range partitioning according to RP*n and RP*c SDDS schemes. Your application sees an RP* file as a familiar B-tree. Main addition to Version 1.0 is that you can now also back up the SDDS files on the disk. The backup uses algebraic signatures, to write back only the data that was modified since the last backup. The network level should support the multicast. It is usually the default for a local Windows network.
SDDS2005 provides all functions of SDDS 2004 and adds on some new capabilities. In particular, the client may encode/decode the records, using the technique of the cumulative signatures. The stored records are then protected against the incidental viewing at the servers. The encoding further allows for the parallel/distributed non key string search possibly faster than it could be on the original data. The string search capabilities of SDDS-2005 are the: prefix search, string search, longest common prefix search and the longest common string search.

This section describes the application interface of SDDS-2005 Client Service. Currently, there are interfaces between SDDS-2005 and the applications programs in *C* and *C++* on *Microsoft Visual C++ 6.0* platform.

These interfaces can be used by system developers that need to access to SDDS-2005 internal primitives. User applications use the local *SDDS-2005* Client Service as any Windows service. Several applications can access the same SDDS-2005 Client Service simultaneously as in the client-server communication model. Each Client Service can be used simultaneously by up to 10 applications.

This interface uses the LPC (Local Procedure Call) mechanism. All the *.h* and *.obj* files are in the directory \Calling Interface\. All the examples codes presented in what follows are from the SDDS-2005 Test Application V 1.0.

Table of contents

1- The Calling Process :

You need the following sources files in the directory \Calling Interface\ to call SDDS-2005 from your applications: *interfaceSDDS.h*, *DataBloc.h* and *SDDSFunctions.h*. We should include these files in our application.

In addition, we need the following files in the directory \Calling Interface\:

- InterfaceSDDS_c.obj, which contains the LPC client definitions of *SDDS-2005* internal primitives The Microsoft libraries *rpcns4.lib* and *rpcrt4.lib*, which provide the LPC mechanism. These file belong to the *Microsoft Visual C++ 6.0* libraries.

Data pass between the application and SDDS-2005 Client Service through data structures implemented with memory-mapped files. Non-key data is limited to max 100 bytes in V 1.0.
In order to application access theses structures, you should specify the installation directory of the SDDS-2005 Client Service. This directory is specified by the variable "*WorkingDirectory*" in the file *DataBloc.h*. By default WorkingDirectory = " ".

For instance if the SDDS-2005 Client Service is installed in C:\Program File\SDDS-2005\SDDS-2005 Application & Client Service, then you should declare:
WorkingDirectory = " *C:\\Program File\\SDDS-2005\\SDDS-2005 Application & Client Service* ".

The different SDDS-2005 interface functions we described below are in file *SDDSFunctions.h*. Each function returns in particular specific error codes.

2- Connections

The application starts with request for the connection between to SDDS-2005 Client Service. This is done through the function

```
int SDDS_Connect();
```

The **Connect** function returns the following codes:

Return Codes	
-3	*
-2	*
-1	*
0	Failure, the service doesn't respond.
1	Success, The connection was created.
2	Failure, the maximum number of supported applications is reached.
3	*

To terminate the connection call the function: .`int SDDS_Disconnect();`

Return Codes	
-3	*
-2	*
-1	*
0	Failure.
1	Success.
2	*
3	*

Example 1:

The following code connects the application to the SDDS-2005 Client.

```
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <rpc.h>
#include <rpcndr.h>

#include "interfaceSDDS.h"
#include "DataBloc.c"
#include "SDDSFunctions.c"

main(void)     {

       if (SDDS_Connect() == 1)      {

                      /* Others codes*/
              …

              if (SDDS_Disconnect() == 0)   {
               printf("\n SDDS_Disconnect: System error. Aborting…\n") ;
                 }
          }
              else    {
       printf("\nSDDS_Connect: System error. Aborting…\n") ;
       }
       }
```

3- File Creation

The following function creates a new SDDS file with the name *FileName*. The file name is any text without spaces up to 15 characters. The variable named *BucketSize* represents the maximum number of records four one bucket. This size is limited only by the RAM size. For a bucket of b records, you need [(1 + (41*b/80,000)) + (b/50)*16,080] bytes of RAM. The variable named *MaximalKey* represents the higher key of the entire SDDS file. We suppose that the minimal key is zero. Finally, *FirstServerAddress* is the address of the starting server of the SDDS file. Initially, its interval is [*0, MaximalKey*].

```
int SDDS_FileCreation(char *FileName,
                      long BucketSize,
                      long MaximalKey,
                      char *FirstServerAddress);
```

Example 2:

```
…
char FileName[15],
     FirstServerAddress[16];
long MaximalKey,
     BucketSize;
int Result;

printf("\nFile name (max string[15]):");          scanf("%s", FileName);
printf("\nMaximal key : ");                       scanf("%d", &MaximalKey);
printf("\nBucket size (min: 100 records): ");     scanf("%d", &BucketSize);
printf("\nFirst serveur address : ");             scanf("%s", FirstServerAddress);

Result = SDDS_FileCreation(FileName,
                           BucketSize,
                           MaximalKey,
                           FirstServerAddress);
```

…

Return Codes	
-3	Maximal Key < Minimal Key.
-2	The bucket size is too small (min: 100 records).
-1	The file name length is too long.
0	The file name is already used.
1	Success, the file was created.
2	No available bucket on this server.
3	No response.

4- File Manipulation

The following interface show different functions in available in SDDS 2005.

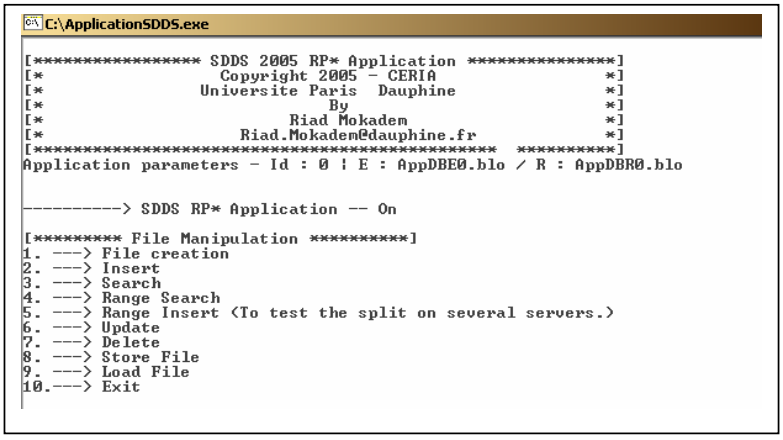


Figure A- 2 : Appplication Interface in Client.

4.1 *Key Insert*

This function sends a key insert request. You must specify the SDDS type you want to use to perform the request in *SDDSType*. For RP*$_n$, set this variable to *SDDSType* = 0, and for RP*$_c$, set it to *SDDSType* = 1. In SDDS2005, data are encoded at clients and then, sent to servers.

```
int SDDS_KeyInsert(char *FileName,
                   unsigned short SDDSType,
                   long Key,
                   unsigned short DataSize,
                   LPVOID Data);
```

Codes returned

-3	*
-2	The data size is too big.
-1	The file name length is too long.
0	The key is already in the file.
1	Success, the key is inserted.
2	*
3	No response.
10	This file doesn't exist.

The following code creates new record with the key you interactively provide in the test application.

Example 3:

```
char FileName[16],
     Data[TailleMaxDonnee];
long Key;
unsigned short DataSize,
     SDDSType;
int Result;

printf("\nFile name (max string[15]):");          scanf("%s", FileName);
printf("\nKey : ");                               scanf("%d", &Key);
printf("\nSDDS type (0 = RP*n / 1 = RP*cu) : ");  scanf("%d", &SDDSType);
printf("\nData (max : 100 bytes) : ");            scanf("%s", Data);
```

```
DataSize = strlen(Data)+1;
Result = SDDS_KeyInsert(FileName,
                        SDDSType,
                        Key,
                        DataSize,
                        Data);
```

Example of inserting data "*universiteparisdauphinetechnologie*" in file m at record 42:

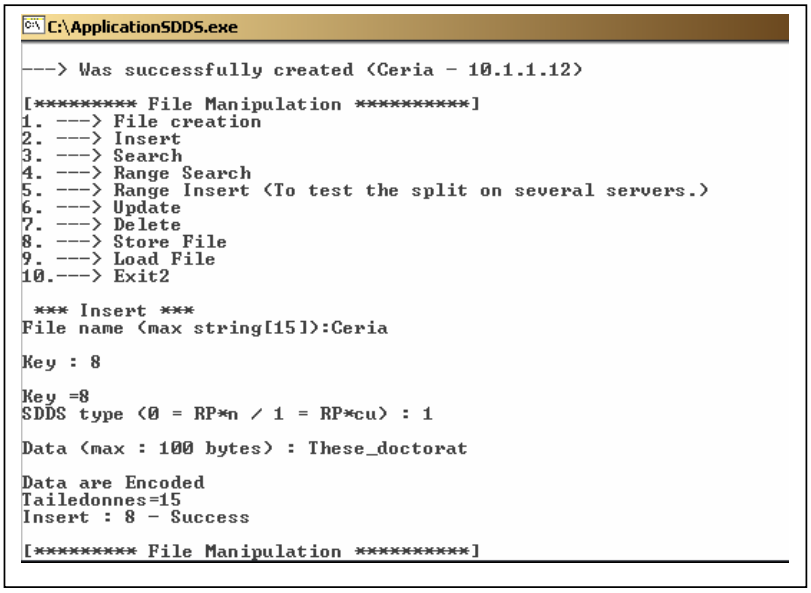


Figure A- 3 : Insertion of records in SDDS-2005

4.2 Key Search

This function sends a key search request. You must specify the SDDS type you want to use to perform the request. For RP*$_n$ then *SDDSType* = 0, and for RP*$_c$ put *SDDSType* = 1.

```
int SDDS_KeySearch(char *FileName,
                   unsigned short SDDSType,
                   long Key,
                   unsigned short *DataSize,
                   LPVOID Data);
```

Return Codes	
-3	
-2	
-1	The file name length is too long.
0	Failure, the key was not found.
1	Success, the key was found.
2	*
3	No response.
10	This file doesn't exist.

The following code searches for the record with the key you interactively provide in the Test Application.

Example 4:

```
char FileName[16],
     Data[TailleMaxDonnee];
long Key;
unsigned short DataSize,
               SDDSType;
int Result;

printf("\nFile name (max string[15]):");          scanf("%s", FileName);
printf("\nKey : ");                               scanf("%d", &Key);
printf("\nSDDS type (0 = RP*n / 1 = RP*cu) : ");  scanf("%d", &SDDSType);

Result = SDDS_KeySearch(FileName,
                        SDDSType,
                        Key,
                        &DataSize,
                        Data);
```

4.3 Key Update

This function sends a key update request. You must specify the SDDS type you want to use to perform the request. For RP*$_n$ *SDDSType* = 0, and for RP*$_c$ *SDDSType* = 1. This function is not used by the Test Application V 1.0.

Return Codes

-3	
-2	
-1	The file name length is too long.
0	Failure, the key was not found.
1	Success, the key was found.
2	*
3	No response.
10	This file doesn't exist.

```
int SDDS_KeyUpDate(char *FileName,
    unsigned short SDDSType,
    long Key,
     unsigned short *DataSize,
    LPVOID Data);
```

4.4 Key Delete

This function sends a key search request. You must specify the SDDS type you want to use to perform the request. For RP*$_n$ *SDDSType* = 0, and for RP*$_c$ *SDDSType* = 1. This function is not available under the Test Application V 1.0.

```
int SDDS_KeyDelete (char *FileName,
                    unsigned short SDDSType,
                    long Key,
                    LPVOID Data);
```

Return Codes	
-3	
-2	
-1	The file name length is too long.
0	Failure, the key was not found.
1	Success, the key was found.
2	*
3	No response.
10	This file doesn't exist.

4.5 Range Queries

The function *Range Search* sends a range search for all records with keys in the interval [*MinimalKey, MaximalKey*]. Then, it waits for the end of the request. This one is dealt by the function whose description follows shortly.

```
int SDDS_RangeSearch(char *FileName,
          long MinimalKey,
          long MaximalKey);
```

Return Codes	
-3	*
-2	Maximal Key < Minimal Key.
-1	The file name length is too long.
0	No response
1	Success.
2	*
3	*

The following function receives the data sent by the servers, in reply to a range query. It runs as a reception thread. `void ThreadReceive_RangeRequestResponse (LPVOID IdThreadRIE);`

Example 5:

It should be started before you issue the range query. Test Application uses 4 concurrent reception threads with number *IdThreadRIE* = 0, 1, 2, 3 initialized by loop by the *SDDS_RangeSearch* function before the range search query.

```
void ThreadReceive_RangeRequestResponse (LPVOID IdThreadRIE)
{
                    unsigned short IdThreadReception = (unsigned short)IdThreadRIE,  // The thread id
                        DataSize,
                        Result;
                    BOOL Flag = TRUE;
```

```
                while (Flag)            {
                        ReceptionIntervalleEnreg(IdApplication,         // A specific function which
                                                IdThreadReception,  // receives the data from
                                        0,                      // one server.
                                                0,
                                                &DataSize,
                                                &Result);
                        if (DataSize > 0)       {
                        PrintDataBloc(IdThreadReception,            // Prints the data on the screen.
                                        DataSize);

                        /* Others codes */
                        …                                           // You can manipulate the data.
                        }

                switch(Resultat)        {

                        case 0:     Flag = FALSE;
                                    SetEvent(hThreadFinReception[IdThreadReception]);
                                    break;
                        case 1:
                                    Flag = FALSE;
                                    SetEvent(hThreadFinReception[IdThreadReception]);
                                    break;
                }
        }
}
```

4.6 Range Insert

This function sends *range key insert* request.

```
printf("\n *** Range insert (Random keys) ***");
printf("\nFile name (max string[15]):");    scanf("%s", NomFicher);
printf("\nNumber ok keys to insert (Max : 3,000,000): ");     scanf("%d", &NbCles);
printf("\nSDDS type (0 = RP*n / 1 = RP*cu) : ");scanf("%d", &AlgoSDDS);
printf("\nMode of Insertion (0=The  %d first keys)",NbCles);
printf("\n                (1=The first %d peer keys 2,4-->%d)",NbCles,NbCles*2);
printf("\n                (2=The %d keys begining at the last key)",NbCles);

scanf("%d", &choix_insertion);
depart=0;
if(choix_insertion==2) {        printf("\nKey of begin insertion:");
                                scanf("%d",&t);
                                depart=t;
                }
```

4.7 Non Key Field search

This function sends a data search request (String Matching fonction).

```
int SDDS_DataSearch(char *FileName,
                    unsigned short SDDSType,
                    unsigned short SearchType,
                    unsigned short *DataSize, LPVOID Data);
                         printf("\n *** Search ***");
                    printf("\nFile name (max string[15]):");                    scanf("%s",
NomFicher);
                    printf("\nSDDS type (0 = RP*n / 1 = RP*cu) : ");            scanf("%d",
&AlgoSDDS);
                    printf("\nSearch of:");
                    printf("   \n 0-Key
\n 1- Complete String (algebraic)
                \n  2-Longest Common String                    […....New
\n  3-Longest Prefix                        SDDS 2005
\n 4- Prefix:
\n 5-String (cumulative sign)
\n 6- n-Gramme Based String Search");    ……New ]
                    scanf("%d", &search);
```

It sends a signature of partial or complete string to search. There are also possibilities to search Prefix or Strings in records of the data bucket. It is also possible to search the longest prefix common or longest string common, we should enter this prefix or string.. You must specify the SDDS type you want to use to perform the request. For RP*$_n$ *SDDSType* = 0, and for RP*$_c$ *SDDSType* = 1. You must also specify search type (By key, string, prefix, longest prefix common or longest string common).

Example of running:

File Name: Ceria

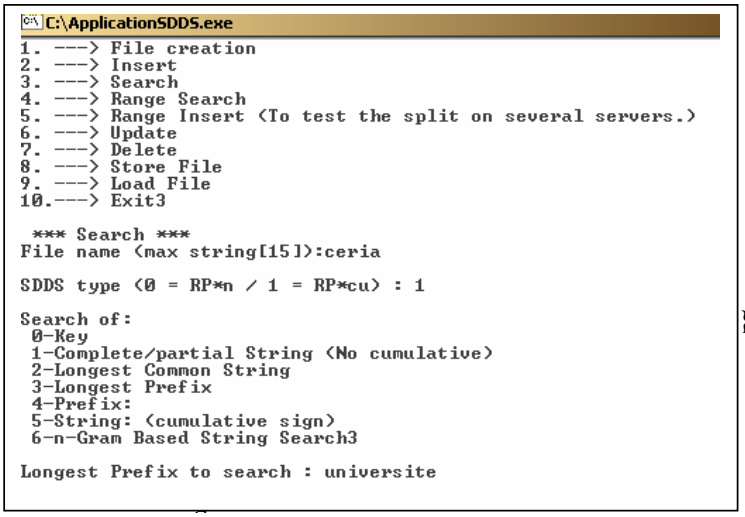
SDDS Type: 1

Search of:

0-	Key:	**99**
1-	1-Complete String:	***RiadMokadem***
2-	Longest Common String:	***Admokad***
3-	Longest Prefix:	***Riadmo***
4-	Prefix:	***Riad***
5-	String:	***Mokadem***
6-	Partial String (*n*-Gramme)	***Riad***

Return Codes

-3	
-2	
-1	The file name length is too long.
0	Failure, the key was not found.
1	Success, the key was found.
2	*
3	No response.
10	This file doesn't exist.



4.8 File Storage

This function sends a file to store in all the local disk in network. You must specify the SDDS type you want to use to perform the request. For RP*$_n$, *SDDSType* = 0, and for RP*$_c$ *SDDSType* = 1. Only data changed are stored in the next request

```
int SDDS_StoreFile(char *FileName,
                   unsigned short SDDSType,
                   unsigned short *DataSize)
```

Return Codes	
-3	
-2	
-1	The file name length is too long.
0	Failure, the File was not found.
1	Success, the File was found.
2	*
3	No response.
10	This file doesn't exist.

```
printf("\n *** Storage ***");
printf("\nFile name (max string[15]):");          scanf("%s", NomFicher);
```

4.9 File Load:

This function sends a file to load from all disk to fresh memory of servers. You must specify the SDDS type you want to use to perform the request. For RP*$_n$ *SDDSType* = 0, and for RP*$_c$ *SDDSType* = 1. Only data changed are stored in the next request

```
int SDDS_LoadFile(char *FileName,
                  unsigned short SDDSType,
                  unsigned short *DataSize)
```

Return Codes	
-3	
-2	
-1	The file name length is too long.
0	Failure, the File was not found.
1	Success, the File was found.
2	*
3	No response.
10	This file doesn't exist.

```
printf("\n *** Loading File ***");
printf("\nFile name (max string[15]):");          scanf("%s", NomFicher);
```

SDDS-2005 Setup

SDDS-2005 V 1.0 for Windows 2000 Server Readme
(c) Copyright 2005 CERIA, University Paris Dauphine, France. All rights reserved.
http://ceria.dauphine.fr
By Riad Mokadem {Riad.Mokadem@dauphine.fr}
http://ceria.dauphine.fr/SDDS-2005/SDDS-2005.HTM

Thank you for using SDDS-2005. This section contains important information about SDDS-2005.

---- Contents ----

---- Introduction ----

SDDS-2005 is a manager of Scalable Distributed Data Structures (SDDSs) for Windows multicomputers, i.e., is for networks of PCs and WSs. The applications manipulate data through the SDDS client nodes. The data are stored at the SDDS server nodes. Data are in the distributed RAM, to provide the access time almost hundred times faster than to the disk. The data distribution over the server nodes is scalable and transparent for the application.
SDDS-2005 V 1.0 provides the scalable range partitioning according to the RP*$_n$ or RP*$_c$ schemes.
These schemes use the multicasting; hence the communication subsystem should be set up accordingly.
It is usually the default for a local Windows network. Contact your system administrator if in doubt.
For the description of the RP* schemes and of SDDSs in general see the papers in http://ceria.dauphine.fr/

---- SDDS-2005 Components ----

SDDS-2005 V 1.0 is structured into three components. These are the Client, the Server and the Name Server. The components run as Windows 2000 or Windows XP services.

The Server
Each server stores a part of an SDDS in the RAM memory space called bucket. Once installed, the server waits for the SDDS bucket creation requests. These come from the clients creating files, or from overflowing servers performing splits. In version V 1.0, a server can have up to 10 buckets of different SDDSs.

The Client
Once installed as Windows service, it called by the local applications through its application interface. Each client can be used by up to 10 applications. See http://ceria.dauphine.fr/ for the description of the interface.

The Name Server
It provides a naming service. It contains all the existing file names. The creation of a new file is refused if the name is already in use.

Only one instance of any component can be run on one machine. We recommend installing each component at a different machine. It is however possible to install the client and one server at the same machine but necessary client ad application in same machine. Likewise one may install the Name server at any machine with a server or a client.

---- Installing SDDS-2005 ----

System Requirements:

Windows 2000 sever (Intel) operating system (recommended) or Windows XP.
Pentium Pentium III or higher(recommended)
Minimum of 64 megabytes of RAM (more is better)
50 megabytes of free disk space
Internet connexion.

Installation Procedure

Download and Install each component. There is a dedicated setup procedure for each component.

The Client

To install the Client, download Client SDDS module. Open the \install A and C\ directory. Click at SETUP icon. Follow the instructions at the screen.
The Client setup installs also the Test Application. This application lets you to interactively verify that all the components work properly. Run it only after installing all the components and after starting from the Windows Start Menu all the SDDS services (the client, at least one server, and the name server). Through the Test Application, you can create an SDDS file, insert a record (data are encoded in severs)[New SDDS2005], insert a batch of records to test the splits, update or insert record, perform a

key or string search (prefix, string search, longest common prefix or string search) [New SDDS2005], or a range query, store or load file. The test data disappear once you stop the SDDS server(s).

The Server

To install the Server, download Server SDDS module. Open the \install S\ directory. Click at SETUP icon. Follow the instructions at the screen.

The Name Server

To install the Client, download name server SDDS module. Open the \install NS\ directory. Click at SETUP icon. Follow the instructions at the screen.

---- Running SDDS-2005 ----

Before you begin, ensure that your TCP/IP is properly configured at all the machines you plan to use. This includes the multicasting capabilities. Furthermore, SDDS-2005 components use some fixed port numbers which should not be used by other applications at the system. In doubt, contact your system administrator. The SDDS-2005 specific ports are as follows.

SDDS-2005 Server:
Sending port: 7201
Receiving Port: 7200
Split port
Waiting split request: 7100
Data transfer: 7150
SDDS-2005 Client:
Sending port: 7500
Receiving Port: 7501 for key request / 7506 for range request
SDDS-2005 Name Server:
Sending port: 8001
Receiving Port: 8000

To launch SDDS-2005 proceed as follows:

1. Start at least one SDDS-2005 Data Server. Start several Servers if splits may occur. Notice that an SDDS-2005 file can scale only at the machines with the active Servers;
2. Start one SDDS-2005 Client copied to C:\ .
3. Start the SDDS-2005 Application copied also in C:\. Play with. (Application must be start in same computer of client and run after client)
4. Start the SDDS-2005 Name Service.

Remarks:

- When the SDDS2005 application, client and name server as well as server are running on the same computer, this machine must be connected to a local network to allow a multicast redirection.
- When application and client are running in machine having Windows XP as operating system, Server and name server must be run in other machine.

---- SDDS-2005 V 1.0 files characteristics ----

File name: any text without spaces up to 15 characters.
Keys : numeric [1; 100,000,000].
Non-key data size: max 64 KB
Bucket capacity: the minimal size is 100 records, limited only by the RAM size. For a bucket of b records, you need [(1 + (41*b/80,000)) + (b/50)*16,080] bytes of RAM. Notice that you need 68 MB of RAM for the Windows 2000 system and the SDDS server itself.

Problems. Send email to *riad.mokadem@dauphine.fr*

Organisation des Programmes de SDDS-2005 V 1.0

Riad Mokadem
Riad.Mokadem@dauphine.fr
CERIA Université Paris 9 Dauphine Place du Mal. De Lattre de Tassigny,
75775 Paris. Cedex 16, France

Serveur

Répertoire	: ServerSDDSService
Programme principal	: ServerSDDSService.c
Threads de travail	: ServerRPInit.c, WThread.c
File d'attente	: FMessages.c
Format des messages	: FormatMessages.c
Gestion des acks	: FAck.c
Eclatement	: FREclate.c, FSEclate.c,GREclate.c, GSEclate.c
Requêtes simples	: ReqSimples.c
Requêtes à intervalles	: ReqIntervalle.c
Gestion des cases	: FCase.c, FFeuilles.c, FNiveauFile.c, FNoeuds.c, RefFile.c
Haute-disponibilité	: RPrs*.c

Client

Répertoire	: ClientSDDSService
Programme principal	: ClientSDDSService.c
Threads de travail	: ClientRPIni.c
File d'attente	: FMessages.c
Format des messages	: FormatMessages.c
Requêtes à intervalles	: Frange.c, FOffreServeur.c, FReponse.c, BRPRange.c
Interface application	: FonctInterface*.c
Contrôle du flux	: FReprise.c, FZLibre.c
Haute-disponibilité	: RPrs*.c

Application Test

Répertoire	: ApplicationSDDS
Programme principal	: ApplicationSDDS.c
Interface avec le client	: InterfaceSDDS*.*
Requêtes à intervalle	: RangeRecep.c

Serveur de Noms

Répertoire	: Name Server
Programme principal	: NameServerRP.c
Threads de travail	: NServerInit.c
File d'attente	: FMessages.c
Format des messages	: FormatMessagesNS.c
Traitement des requêtes	: TraiterReq.c
Traitements des acks	: FAck.c
Haute-disponibilité	: HA-RP-NServer.c

Articles de Recherche

Dans ce qui suit, nous mettons à disposition des lecteurs, la communauté anglophone notamment, quelque uns de nos articles publiés dans des conférences internationales.

R Mokadem, W Litwin, Thomas Schwarz.: "*Disk Backup Through Algebraic Signatures in Scalable and Distributed Data Structures*", Proceedings of the Fifth Workshop on Distributed Data and Structures, Thessaloniki, June 2003 (WDAS 2003).

W Litwin, R.Mokadem & Th.Schwarz. "*Cumulative Algebraic Signatures for fast string search. Protection against incidental viewing and corruption of Data in an SDDS*". Springer in LNCS series. Proceeding DBISP2P 2005)

W Litwin, R.Mokadem & Th.Schwarz "*Pre-computed Algebraic Signatures for faster string search. Protection against incidental viewing and corruption of Data in an SDDS*". Proceeding of third international workshop and co-located with the 31st international conference on Very Large Data Bases (DBISP2P 2005). Trondheim, Norway. August 2005.

R.Mokadem, W Litwin "*String-matching and update through Algebraic Signatures in Scalable Distributed Data Structures*". 3rd International Workshop on "*P2P Data Management, Security and Trust"* (PDMST'06) Krakow, Poland 4 - 8 September 2006.

W Litwin R, Mokadem, T Schwarz. & P Rigaux. '*Pattern Matching Using Cumulative Algebraic Signatures and n-gram Sampling'*. DEXA 2006. Seventeenth International Conference on Databases and Expert System Application. W05 - GLOBE '06 (Grid and Peer-to-Peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems). Krakow, Poland 4 - 8 September 2006.

R.Mokadem. '*Fast String Search Using n-Gram Sampling and Cumulative Algebraic Signatures: Preliminary Results'*. IEEE/ ACM International Conference on Signal/ Image technology and Internet/ Based Systems (STIS'06). To appear. Tunisia Dec 2006.

W Litwin, R Mokadem, S Sahri. "*Virtual Repository For e-Gov Life Event Document*". Communication Proceedings of the Fifth International EGOV Conference. Poland Sept 2006.