Centre de Recherche en Informatique Appliquée

MASTER RECHERCHE, INFORMATIQUE SYSTEMES INTELLIGENTS
DIRECTION DE RECHERCHE BASE DE DONNEES

Thème

« LH*rsP2P : une Nouvelle Structure de Données Distribuée et Scalable, pour un environnement Pair à Pair »

Présenté par
M. H.YAKOUBEN

Proposé et dirigé par le
Pr. W.LITWIN

Promotion 2005/2006

Remerciements

Je remercie M LITWIN, pour son encadrement, ainsi que pour sa disponibilité et ses remarques constructives qui m'ont été très utiles tout au long de mon stage et pour la confiance qu'il m'a toujours témoigné. Qu'il soit ici assuré de ma profonde gratitude et de mon très grand respect.

Je tiens à remercier, Sorror et Riad pour leurs conseils et leurs aides précieuses, Melle R.MOUSSA pour les explications sur son prototype et sans omettre Samira pour avoir accepter de lire ce mémoire.

Mes remerciements vont aussi au Professeur Michel SCHOLL pour avoir accepter la responsabilité du stage et le Professeur Thomas SCHWARZ pour ces conseils sur le 'Churn'.

Je tiens à remercier Mm JOMIER, M. RIGAUX et LITWIN, membres du jury, pour l'honneur qu'ils m'ont fait en acceptant de juger notre travail. Je remercie également tous les professeurs du Master ISI, dont j'ai suivi les cours avec grand intérêt.

Mes remerciements vont également à Madame Suzanne PINSON, responsable du Master ISI et Madame Geneviève JOMIER, responsable de la direction de recherche 'Base de Données' pour leur suivi attentif de mes études durant toute la durée du Master.

Je tiens à remercier aussi tout ceux qui ont contribué de près ou de loin à l'élaboration de ce travail.

H.YAKOUBEN

Dédicaces

Je dédie ce modeste travail à :
Ma chère famille
Et Mes amis

Sommaire

Chapitre I. Les Structures de Données Distribuées et Scalables

*Chapitre II. LH*P2P : Hachage Linéaire Distribué et Scalable Pair à Pair*

Chapitre III : Architecture Fonctionnelle du Système

Chapitre IV. Domaine d'Application

Table des figures

Introduction générale

Actuellement, nous assistons à une globalisation de l'interconnexion des réseaux. Internet en est l'exemple le plus pertinent. Cette nouvelle tendance a rendu pertinent le concept de multi-ordinateur. Nous avons désigné ainsi une collection de stations de travail et de PCs, dits nœuds ou sites, interconnectés et organisés d'une manière qui mettrait en commun leur capacité de stockage et de calcul. Comme s'ils formaient ensemble un grand, voir un gigantesque ordinateur.

Le concept de multi-ordinateur continue d'évoluer. Ces dernières années on s'est, d'une part, orienté vers des architectures multi-ordinateur dites pair à pair (ang : peer to peer, P2P). Un système pair à pair est un multi-ordinateur à noeuds relativement similaires et particulièrement autonomes du point de vue de leur administration. Il en suit une disponibilité relativement faible d'un nœud individuel. Une autre direction est constituée par les systèmes en grille (ang. grids). Il s'agit de multi-ordinateurs à noeuds moins autonomes, mais plus disponibles par contre.

Les multi-ordinateurs exigent de nouvelles organisations de données. Celles-ci doivent répondre à de nombreux impératifs. Il s'agit du stockage de grands volumes, d'adressage décentralisé, de scalabilité, de haute disponibilité et de sécurité accrue. Pour répondre à ces exigences, une nouvelle classe de structures de donnée dite Structures de Données Distribuées et Scalables (SDDSs) [LSN93] a été développée. Les recherches sur les SDDSs ont concerné divers types d'adressage, les prototypes de systèmes de fichiers distribués (SFD) ainsi que les systèmes de gestions de bases de données (SGBD).

Les données d'une SDDS, d'un fichier SDDS notamment, sont supposées réparties sur les sites de stockage du multi-ordinateur dits serveurs. Pour leur traitement, en général ces données sont dans la mémoire (RAM) distribuée. Elles sont alors dix à cent fois plus rapidement accessibles qu'à partir de disques. Les applications manipulent une SDDS à partir de sites dits clients. Ceux-ci gèrent l'accès, mais ne stockent pas de données de la SDDS. Certains sites assurent les deux fonctions à la fois, ils sont dits les sites pairs. Une SDDS s'adapte dynamiquement à l'accroissement du volume des données. Le fichier s'étend de manière progressive d'un seul site serveur ou pair à, théoriquement, un nombre illimité de tels sites. Le placement de données d'une SDDS et son évolution sont transparents pour les applications. Les utilisateurs ou les applications peuvent manipuler les données d'une SDDS via une interface offerte par un client. On peut faire des opérations similaires à celles sur un fichier centralisé. Ainsi il peut s'agir de recherches par clé, de requêtes par intervalle, d'insertions, de mises à jour…etc.

Plusieurs SDDS sont connues. Citons d'une part : LH*[LSN93] et RP* [LSN94] parmi les premières. Citons aussi CHORD [Stoica01], BATON [JOV05], VBI-Tree [JOV0P6], pour une architecture pair à pair, ainsi que LH*rs, [LMS02], pour la haute disponibilité parmi les propositions les plus récentes. Une propriété générale de toutes les SDDS est que l'adressage à partir d'une clé peut nécessiter des messages de renvois entre les serveurs/pairs. LH* garantit deux renvois en maximum, quelque soit le nombre de serveurs de la SDDS. Il est jusque-là la SDDS la plus rapide dans ce sens. Ceci, contre un coût de l'ordre *O(Log N)* messages pour les autres organisations pair à pair citées.

Dans le but de réduire le nombre de renvois nous proposons d'abord une nouvelle SDDS, appelée LH*P2P. Nous élaborons ensuite la conception et l'implémentation d'une variante à haute disponibilité dite LH*rsP2P. Il s'agit de la généralisation correspondante de LH*rs. Les deux SDDSs réduisent le nombre de renvois à *un seul* dans un environnement pair à pair. Un résultat impossible à améliorer car la borne de '*zéro renvoi'* conduit à une architecture centralisée.

Pour présenter notre travail, nous avons organisé notre mémoire comme suit :

Dans le premier chapitre, nous présentons plus en profondeur l'état de l'art des SDDS. Nous présentons les SDDS en général et LH* ainsi que LH*rs en particulier. Le chapitre suivant présente l'algorithmique de LH* P2P et de LH*rs P2P. Ensuite, nous décrivons l'architecture fonctionnelle de notre système implémentant LH*rsP2P. Cette implémentation utilise le prototype existant de LH*rs, [LMS04]. Puis, nous présentons une application potentielle de Lh*rsP2P à la gestion de documents dans le cadre du projet de la communauté économique européenne (CEE) 'eGov'. Nous terminerons par une conclusion générale et les perspectives.

Les Structures de Données Distribuées et Scalables (SDDS)

Au cours de ces dernières années, la performance des réseaux a connu des améliorations importantes avec des débits de plus en plus élevés, conséquence de l'augmentation de la vitesse des CPU et de la taille des RAM. Cependant les disques durs, limités par des contraintes mécaniques n'ont pas atteint le même succès du point de vue temps d'accès. De là, vient l'idée d'exploiter les performances des multi ordinateurs en initiant beaucoup de projets au niveau de la recherche en informatique, en particulier sur les structures de données distribuées.

La recherche sur les multi-ordinateurs a montré notamment le besoin de nouveaux algorithmes et structures de données. Les Structures de Données Scalables et Distribuées ou **SDDS** (***S**calable and **D**istributed **D**ata **S**tructures*) ont été introduites pour répondre à ce besoin [LNS93a, LNS93b].

Les SDDS assurent des temps d'accès beaucoup plus courts que les performances d'accès aux données stockées sur les disques. Ces nouvelles structures disposent, de plus du traitement parallèle, d'une capacité de stockage potentiellement illimitée. Ces caractéristiques assurent aux **SDDS** des performances de traitement supérieures à celles des structures de données traditionnelles.

Ce chapitre commence par une présentation des multi-ordinateurs. Ensuite il introduit les SDDS et leurs caractéristiques. Nous nous intéresserons particulièrement à une SDDS appelée LH*rs [LMS04], sur laquelle repose en partie notre travail.

I. Les Structure de données distribuées et scalables

I.1. Les multi-ordinateurs

Un *multi-ordinateur* est une collection d'ordinateurs sans mémoire partagée reliés par un réseau de type LAN ou WAN (ang : Local Area Network, Wide Area Network) ou un bus [T95]. La puissance théorique cumulée des ressources en calcul et en mémoire d'un multi-ordinateur est impossible à atteindre pour un ordinateur traditionnel. Cette perspective a favorisé l'apparition de plusieurs projets de recherche autour de la conception et l'application d'un multi-ordinateur. L'un des axes de recherche majeur est, la construction de systèmes de gestion de fichiers scalables et distribués, résidant pour le traitement entièrement dans la RAM distribuée du multi-ordinateur. L'évolution du matériel et des réseaux, notamment la vitesse d'accès à la mémoire distante par rapport au disque local (cf. Figure 1) encourage cette tendance. Les applications potentielles sont les SGBD, les calculs hautes performances et les programmes à haute disponibilité ou le temps réel.

La conception d'un multi-ordinateur nécessite la réfection de logiciels systèmes. Il sont à (re)faire, notamment les SGF (Systèmes de Gestion de Fichiers).

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

Figure 1. Estimation des temps d'accès aux données.

I.2. Les principes architecturaux d'un SDDS

Les SDDS constituent une nouvelle famille de structures de données, basées sur le modèle client/serveur**.** Elles stockent les données dans des cases (ang : Bucket) sur des sites appelés *serveurs*, d'autre sites dit *clients* y accèdent. Les sites clients gardent des paramètres pour le calcul de l'adresse des fichiers sur les sites serveurs. Ces paramètres constituent l'image du client sur le fichier SDDS. Toute application ou un usager d'une SDDS utilise pour l'accès aux données l'interface offerte par un client.

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

Il est utile que les données d'une SDDS résident pour le traitement en mémoire vive distribuée du multi-ordinateur. En effet, le temps d'accès des données en mémoire vive distribuée est bien plus court que celui des données stockées sur disque. Plusieurs SDDSs ont été conçu sur ce principe.

Il a été démontré, notamment par les prototypes, que toutes ces caractéristiques peuvent assurer à une SDDS des performances supérieures à celles des structures de données classiques

L'architecture de toute SDDS se base également sur les principes suivants, [LNS94, KLR94] :

- Les fichiers SDDS n'ont pas de répertoire central d'accès, cela évite d'avoir des points d'accumulation, ce qui engendrerait des goulots d'étranglement et dégraderait les temps d'accès aux fichiers.
- Un fichier SDDS est étendu suite à des insertions d'une manière incrémentale et transparente pour l'application (client). Il débute généralement sur un seul site (serveur) initial de stockage jusqu'à la surcharge de ce dernier. le fichier est alors étendu par un éclatement qui transfert la moitié des données vers un autre site de stockage. Le fichier en question peut être étendu progressivement à un nombre théoriquement illimité de sites (serveurs).
- Les serveurs sont utilisés par des applications autonomes appelées "client". Chaque client supporte le logiciel propre aux SDDS et gère notamment sa propre image de la structure du fichier SDDS. Puisque les modifications du schéma de la structure du fichier SDDS, dues aux éclatements, ne sont pas diffusées d'une manière synchrone aux clients, cette image peut être inexacte. De ce fait, un client est susceptible de faire des erreurs d'adressage et d'envoyer une requête à un serveur incorrect.
- Chaque serveur est capable de détecter une erreur d'adressage le concernant. Si une erreur est détectée, la requête est redirigée vers un autre serveur susceptible d'être le correct. Mais ceci n'est pas toujours le cas et d'autres redirections peuvent survenir. La conception d'une SDDS doit garantir toutefois que le nombre de redirections est toujours réduit. L'un des serveurs participant dans le processus de redirection doit, pour cela, envoyer un message correctif noté **IAM** (ang. ***I**mage **A**djustment **M**essage*) au client ayant fait l'erreur d'adressage. C'est la raison pour laquelle le client ajuste son image. Une SDDS efficace doit produire un minimum d'erreurs d'adressage et de redirections en garantissant qu'au moins la même erreur ne se reproduira plus.

I.3. Caractéristiques d'une SDDS

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

I.3.1. La scalabilité

La scalabilité est la possibilité de gérer un fichier grandissant et accédé par davantage de client avec les mêmes performances d'accès [G93, G99]. Deux facteurs principaux caractérisent la scalabilité d'un système, il s'agit de :

- *Facteur de rapidité (ang : Speed-up)* : permet de mesurer la diminution du temps de réponse en augmentant le nombre de nœuds pour l'exécution d'une requête sur un même volume de données. Autrement dit, si la capacité augmente d'un facteur de *n*, alors dans un système scalable, le temps de réponse d'une requête diminue d'un facteur de *n,* (cf. Figure 2).
- *Facteur d'échelle (ang : scale-up),* mesure la conservation du temps de réponse d'une requête pour une augmentation proportionnelle de la taille de la base de données et des capacités de la configuration. Autrement dit, si la taille d'une base de données

augmente d'un facteur de *n*, alors il suffit d'augmenter la capacité de la configuration (cf. Figure 3).

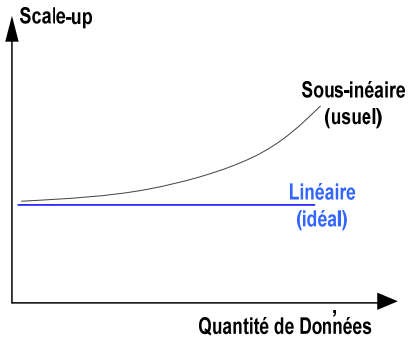


Figure 2. Facteur d'échelle

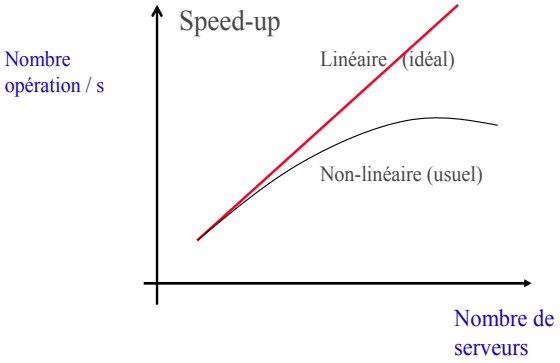


Figure 3. Facteur de rapidité

I.3.2. La disponibilité

Lorsqu'une structure de données (fichier) s'étend sur plusieurs serveurs, la probabilité qu'elle ne soit pas entièrement disponible à un moment donné augmente. Par exemple, si *n* est le nombre de serveurs, et *p* la probabilité pour qu'un serveur soit en service, alors la probabilité *P(n)* pour que la structure entièrement soit disponible est P^n . Si $P = 0.98$ et $n = 10$, alors $P(n) = 0.81$. Si $n = 100$ alors $P(n) = 0.00000000168$, c'est à dire zéro [LRR97]. Un moyen de résoudre le problème de la disponibilité est d'ajouter des serveurs de parité à la structure distribuée [LS00].

Le problème de disponibilité concerne aussi les architectures pair à pair.

I.3.2.1. Le '*Churn*'

L'une des caractéristiques d'un environnement pair à pair est la dynamique de l'adjonction et départ de pairs. En effet l'arrivée et départ de pairs est un phénomène non déterministe, donc aléatoire, ce phénomène est aussi appelé '*Churn'*. Toute application pair à pair réelle (indépendamment de la conception théorique de l'application) est confronté au '*Churn'*. L'efficacité d'une application pair à pair tient en compte la manière dont celle-ci gère le '*churn*' pour garantir la disponibilité des informations après le départ d'un pair.

I.4. Classification des SDDS

Plusieurs recherches ont été initiées dans le domaine des SDDS. Elles consistent, essentiellement, à prendre une structure de données classique et essayer de l'adapter aux environnements distribués. C'est le cas, pratiquement, de toutes les SDDS proposées jusqu'à présent.

Selon l'organisation interne des données et le mécanisme qui permet d'y accéder, les SDDS peuvent être classées en plusieurs catégories, comme le montre Figure 4 ci-après. Nous allons discuter cette classification maintenant.

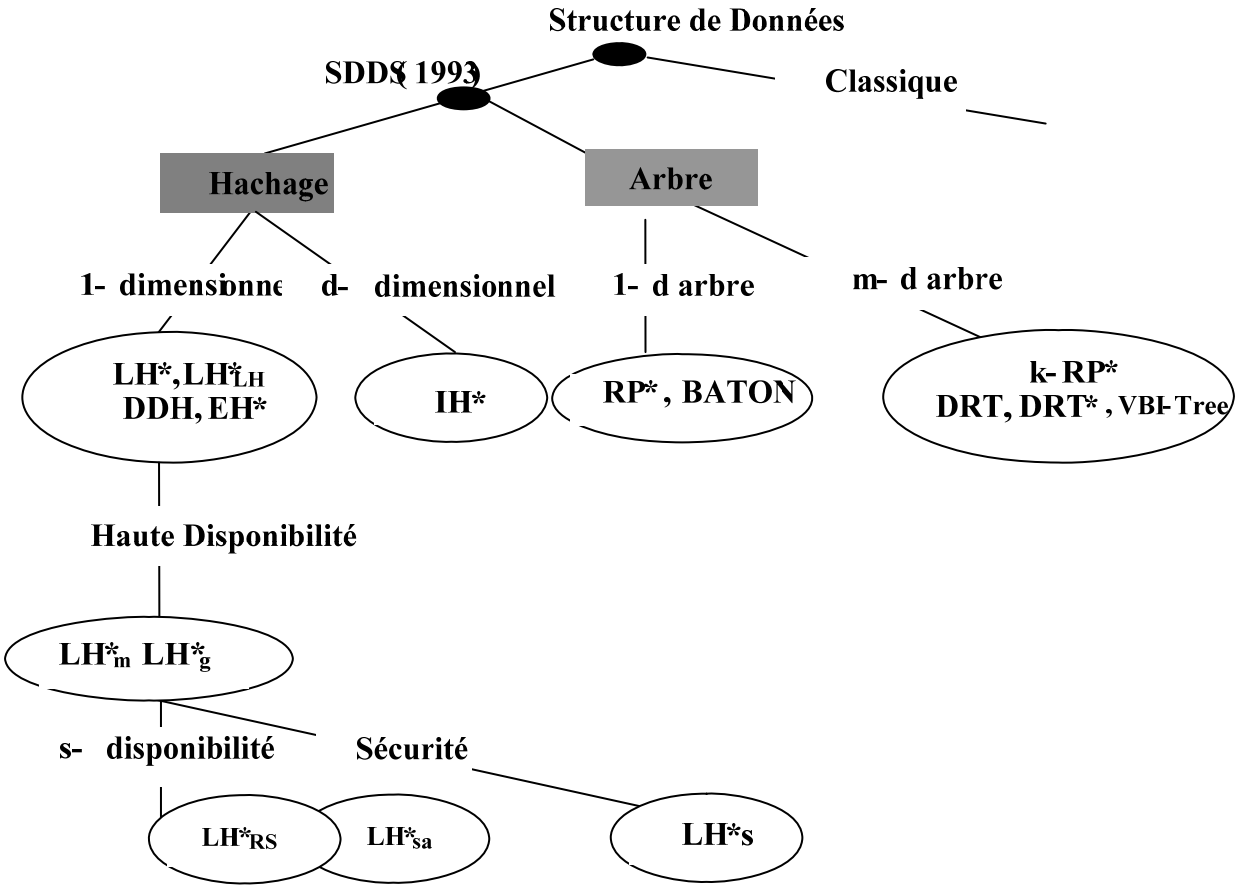


Figure 4 Classification des SDDS

I.4.1. SDDS basées sur les arbres

Les structures de données ordonnées telles que les B-arbres sont plus rapides, si le fichier doit supporter des requêtes par intervalle, un parcours transversal des enregistrements ou une rechercher de l'enregistrement le plus proche (ang : nearest, neighbor search). Pour cela, une famille de SDDS pour les fichiers (mono-clés) ordonnés est proposée, appelée RP* pour « Distributed and Scalable Range Partitioning » [LNS94]**,** dite aussi partitionnement par intervalle. Cette famille est constituée de plusieurs variantes : **RP*$_N$** (No index), **RP*$_C$** (Client index) et **RP*$_S$** (Server index) [LNS94]**.** Elle a été proposée pour construire des fichiers distribués qui préservent l'ordre des enregistrements.

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

Pour retrouver un enregistrement, un client envoie un message à tous les serveurs potentiels (message en multicast[1]). Chaque serveur ayant reçu le message vérifie si la clé de l'enregistrement demandé appartient à son intervalle. Si oui, l'enregistrement est retourné au client sinon la requête est négligée.

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

Cette variante permet au client d'envoyer les requête en unicast[2]., Le client possède à cette fin un index avec son image du fichier SDDS. Un client peut faire une erreur d'adressage par suite

[1] ***multicast*** *: message destiné à un ensemble de machines qui appartiennent à un même groupe de réseau donné.*

[2] ***unicast*** *: message destiné à une seule machine.*

d'une image inadéquate. Dans ce cas, le serveur recevant la requête la redirige par multicast.celui qui possède l'enregistrement le retourne au client avec un IAM (Image d'Ajustement Message).

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

Cette variante introduit une image du fichier SDDS au niveau même des serveurs. La redirection en cas d'erreur se fait par messages unicast. L'image sur les serveurs est construite sous la forme d'index B-arbre distribué [AWD01]. L'index est paginé et chaque page est stocké dans un serveur séparé. Il existe les serveurs d'index et les serveurs de cases.

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

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

Il a était montré que **DRT** et **RP*** permettent de construire des fichiers plus larges et plus rapides par rapport aux structures de données traditionnelles [LNS96].

RP* est utilisé pour implémenter un système de fichiers distribués et scalable appelé SDDS2005 [LMS03]. Ses principes sont également utilisés pour l'implémentation d'un nouveau système de gestion de bases de données appelé SD-SQL Serveur (ang , Scalable and Distributed SQL Server) [LRS02].

I.4.2. SDDS basées sur le hachage :

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

- **LH*** (*Scalable and Distibuted Linear Hashing)* [LNS93] : premier schéma proposé en 1993; c'est une adaptation du hachage linéaire *(Linear Hashing)* [L80] aux environnements distribués.
- **LH*$_{LH}$** [KLR96] : est une variante de LH*. Ce schéma utilise un premier niveau d'index correspondant à LH*, permettant aux clients d'accéder aux serveurs. Le deuxième niveau d'index correspond à l'organisation interne des cases de données suivant l'algorithme LH.
- **DDH** *(Distributed Dynamic Hashing)* [D93]: c'est une version distribuée du hachage dynamique. DDH permet plus d'autonomie d'éclatement, par l'éclatement immédiat des cases en débordement donc on n'a pas besoin de site coordinateur[1] , ce qui peut être un avantage. La DDH a introduit la notion populaire dite en anglais Distributed Hash Table (DHT).
- **EH*** *(Distributed Extensible hashing)* [HBC97] : c'est une version scalable et distribuée du hachage extensible. EH* permet une bonne utilisation de l'espace de stockage et offre un mécanisme de traitement de requêtes très efficace.

[1] ***Site coordinateur*** *: SC en anglais Split Coordinateur. C'est un serveur particulier chargé de gérer les éclatements et de garder une trace des paramètres réels du fichier.*

- **IH*** *(Distributed Interpolation-based hashing)* [BZ02] : Elle constitue, à la fois, une adaptation du hachage par interpolation proposé par W. Burkhard [Bur83] aux environnements distribués et une introduction de la notion d'ordre et de l'aspect multidimensionnel à la SDDS LH***** de Litwin.

D'autres variantes de LH***** ont été proposées pour assurer la haute disponibilité. Celle-ci assure à une SDDS la continuité de fonctionnement de manière transparente, quand un ou plusieurs de ses sites serveurs tombent en panne. Ces SDDS font généralement appel à certains principes de redondance et de récupération de données. Il s'agit de : *Miroring* pour LH*$_M$ [LN96], la fragmentation (*Striping*) LH*$_S$ [LN96], et le groupement d'enregistrements (*Grouping*) LH*$_G$ [L97] [LMR98], la s-disponibilité tels que LH*rs utilisant Reed-Solomon Code [LS99] et LH*$_{SA}$ (Scalable Availability) [LMR98]. Nous nous intéresserons particulièrement à LH*rs que nous allons décrire un peut plus dans ce qui suit.

I.4.3. Adaptation des structures de données aux environnements pair à pair

Ces dernières années, nous assistons à un retour aux origines de l'Internet, car Internet été conçu comme un système pair à pair (ang : Peer to Peer, P2P). En effet plusieurs applications de partage de fichiers sur le net sont disponibles, nous citons, Gnutella, Kazaa,...etc. Les problèmes de fonctionnement inefficace de ses systèmes sous la charge (an flooding) ont également montré l'utilité de la distribution de structures de données pour construire des fichiers scalables. Nous citons à tire d'exemple CHORD [Stoica01], BATON [JOV05]et VBI-Tree [JOV06]

CHORD

C'est une adaptation des tables de hachage dynamique noté DHT (ang : Distributed Hash Table) [D93] aux environnements pair à pair. L'architecture du réseau pair à pair est un anneau, où chaque pair gère une partie de la DHT. Les clés et les pairs sont hachés sur le même anneau. La fonction de hachage est sur *m* bits. Les clés sont dans l'intervalle [0..2^m-1]. Chaque pair a à un identifiant affecté dans un même intervalle, il y 2^m pairs au plus. Une opération de recherche nécessite *Log N* renvois, pour *N* nœuds du système

BATON (A Balanced Tree Overlay Networks)

C'est l'adaptation des arbres de recherches binaires équilibrés[1] aux environnements pairs à pairs. Le coût de l'opération de mise a jours lorsqu'un nœud quitte le réseau, est de *O* (Log *N*) messages. Le coût d'une opération d'insertion, de recherche, et de suppression d'article est de *O(LogN)* messages au pire des cas.

VBI-Tree (Virtual Binary Index - Tree)

La conception de VBI-Tree est inspirée de BATON. C'est l'adaptation de BATON aux données multidimensionnelles (fichier multi-attributs). Nous distinguons deux types de noeuds dans VBI-Tree, à la différence de BATON, des nœuds feuilles dit de données (ang : data nodes) et des nœuds internes dit de routages (ang : Routing nodes). Idem que BATON la complexité, pour satisfaire une requête d'insertion, rechercher… est de l'ordre logarithmique

II. Principes et architecture fonctionnelle de LH*rs

De toutes les SDDS nous nous intéressons particulièrement à LH*rs [LMS04][M04][LMS05], dont nous donnons ses principes de bases.

[1] Arbre binaire équilibré est un arbre binaire tel que les hauteurs (profondeurs) des deux sous arbres de tout l'arbre diffèrent de 1 au plus. |hauteur (arbre gauche)-(hauteur (arbre droit)| <=1

II.1. Adressage

Le principe d'adressage de LH*rs est le même que celui de LH*, nous utiliseons ce même principe pour l'élaboration de nouvelle SDDS.

Le principe d'adressage de LH*rs est le même que celui de LH*. L'algorithme d'adressage du hachage linéaire ci-dessous permet d'affecter un article ayant pour clé *C*, à une case de donnée d'adresse *a*. $h_i(C) = C \bmod 2^i$ est la fonction de hachage utilisée. Le paramètre *i* désigne le *niveau* du fichier et le paramètre *n* dit *pointeur d'éclatement* pointe la prochaine case à éclater.

a ← hi(C) ;
if a < n then a ← hi+1(C) ;

L'architecture de LH* est composé de trois éléments essentiels : un coordinateur, un serveur de données avec une case de données et un client. Le coordinateur est le seul qui a l'état réel des paramètres d'adressage constitué par les paramètres *(i,n)* du fichier. Il gère aussi les éclatement, qui se font dans un ordre déterministe : 0,0,1,0,1…2^{*i-1} - 1,0,1…

La mise à jour de l'image *(i,n)* est assurée par l'algorithme suivant exécuté à chaque éclatement:

n ← n+1 ;
if n<2^i then n←0 ; i←i+1 ; /* i est le niveau de la fonction de hachage*/

Un client LH* utilise son image notée *(i',n')* pour manipuler le fichier SDDS. L'algorithme utilisé est celui de LH* plus haut, mais appliqué à l'image :

a' ← hi'(C) ;
if a' < n' then a ← $h_{i'+1}$(C) ;

A la réception de la requête du client, le serveur (case) vérifie si celle la lui est bien destiné. Dans le cas contraire le serveur renvoie la requête à un autre serveur LH*. Le nombre de renvois est de deux au maximum (preuve dans [LNS93]). L'algorithme utilise est comme suit :

a' ← hj(C) ;
if a'≠a then accept C
else a" ← hj_{-1}(C) ;
if a"> a and a"<a' then a' ← a";
send C to bucket a';

Nous montrerons l'expansion d'un fichier LH* dans le prochain chapitre. Celle-ci est en effet entièrement reprise par LH* P2P, avec des extensions spécifiques à ce dernier

II.2. Principe de LH*rs

$LH*_{RS}$ est une SDDS à disponibilité scalable. Cette SDDS propose à cette fin un code à correction d'effacements de type de Reed Solomon généralisé.Comme son nom l'indique, $LH*_{rs}$ adresse les enregistrements de données sur les cases de données en utilisant LH*. La couche de

haute disponibilité est assurée par le concept de groupage de cases de données en *groupes de parité* et le calcul de parité par le code proposé. Un *groupe de parité* est formé de *m* cases de données consécutives et de *k* cases de parité. Le groupe est dit alors *k-disponible*, car il peut survivre à la panne de *k* de ses cases. Le concept de *groupes de parité* à l'échelle des cases s'applique au niveau des enregistrements, on parle alors de *segments de parité* par groupe d'enregistrements de données. Notons que, chaque enregistrement de données a un rang *r* qui reflète sa position dans la case de données. Un groupe d'enregistrements de rang *r* contient tous les enregistrements de données ayant le même rang dans le même groupe de parité. Il est muni de *k* enregistrements de parité de clé *r*, chacun dans le serveur de parité distinct. La Figure 5, illustre un groupe de quatre cases de données (*m*=4), et auquel sont rattachées deux cases de parité (*k*=2).

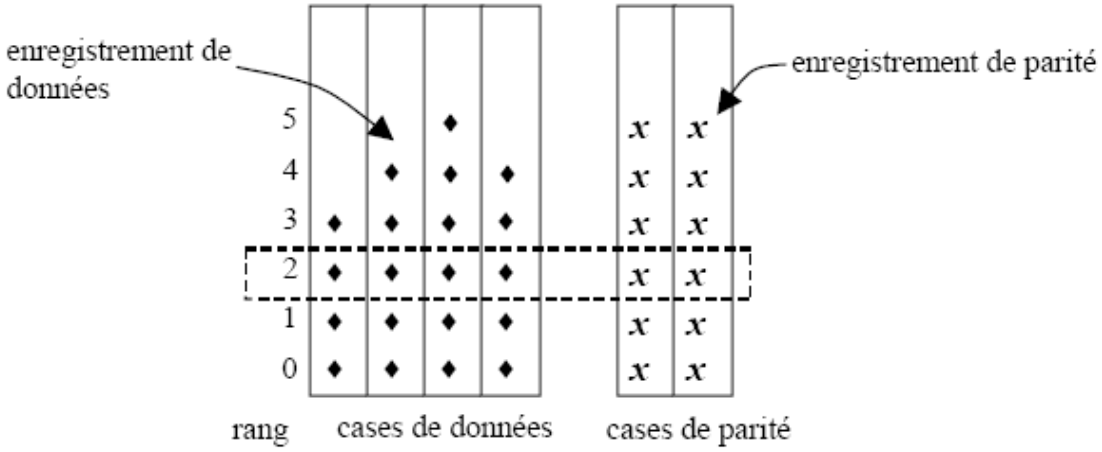


Figure 5. Un groupe de parité d'un fichier LH*rs m=4, k=2

Dans le but de garder une relation entre les enregistrements de données et les enregistrements de parité du même segment de parité, la solution suivante a été proposé dans [LS00]**.** (cf**.** Figure 6**.**a, Figure 6.b)

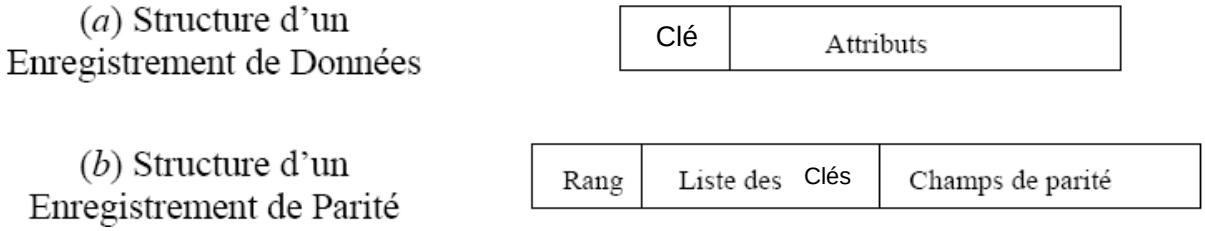


Figure 6. Structure des enregistrements de fichier LH*rs

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

Rang, désigne la clé de l'enregistrement de parité,

Liste des Clés, désigne l'ensemble de *m* clés des enregistrements de données, ayant un rang *Rang* dans les cases auxquelles ils appartiennent.

Champs de parité, ce dernier est calculé, en utilisant le codeproposé, à partir des champs *Attributs* des *m* enregistrements de données, ayant le même rang d'insertion : *Rang*, dans les cases auxquelles ils appartiennent.

III. Architecture système du prototype LH*rs

Dans ce qui suit, nous décrivons l'architecture logiciel du prototype LH*rs (cf. Figure 7). Notons que l'architecture de LH*rs est inspirée du prototype SDDS 2000 [Ben00] qui support la SDDS LH*lh. Nous avons deux principaux éléments qui composent le prototype LH*rs, le client SDDS et le serveur SDDS dit aussi case SDDS. Nous distinguons deux types de serveurs SDDS, les serveurs de données et serveurs de parités comme nous le montre la Figure 7 ci-dessous.

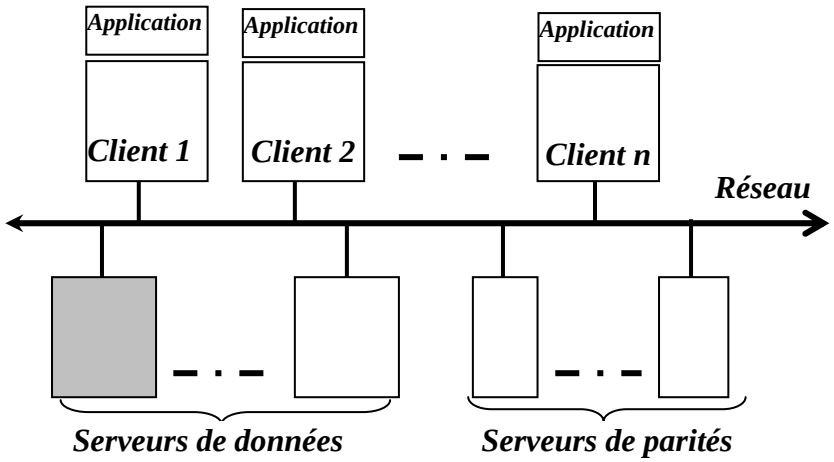


Figure 7. Architecture du prototype LH*rs

L'architecture du prototype LH*rs est basée sur le modèle Client/Serveur. Les différents composants communiquent via les protocole TCP/IP et UDP. Chaque composant a son adresse IP et son port d'écoute/envoi. Notons que le calcul de ports d'ecoute/envoi est uniformisé dans LH*rs. Il dépend du numéro attribué par le coordinateur[1] qui gère les éclatements et la haute disponibilité. Dans ce prototype, le coordinateur gère en plus une case de données. Ainsi il se charge d'attribuer un *numéro d'entité* et un numéro logique à chaque nouveau nœud. Le *numéro d'entité* permet de calculer du port d'écoute/envoi UDP et connexion TCP/IP. Les ports sont calculé par la formule suivante : *port d'écoute UDP* : 6000+(3*c), a pour *port d'envoi UDP* : 6000+(3*c)+1, et a pour *port de connexion TCP/IP* : 6000+(3*c)+2. c est le numéro d'entité de la case.

III.1. Client SDDS

Un client *SDDS* assure les fonctions envoi de requêtes et réception de réponses aux requêtes. Les différentes fonctionnalités d'un client sont assurées par trois processus légers dit *threads*. Il s'agit du *Thread Application*, *Thread Ecoute UDP*, et le *Thread de Traitement de Réponses*. Dans ce qui suit, nous décrivons chacun de ces *threads*.

[1] Coordinateur : serveur spécial qui gère les éclatements et la haute disponibilité

III.1.1. Thread Application

Le *Thread Application* affiche un menu à l'utilisateur, c'est l'interface entre lui et le client SDDS. Le menu permet de manipuler le fichier SDDS par les opérations de bases : insertion, recherche, suppression et autres fonctionnalité telle que la récupération de case.

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

III.1.2. Thread Ecoute UDP

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

III.1.3. Thread de Traitement de Réponses

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

III.2. Serveur

Comme pour une case *SDDS2000*, qu'elle soit de données ou de parité, le serveur fonctionne en mode concurrent, et se base sur la mise en oeuvre du multi-tâches par l'utilisation des *threads*. Les différentes fonctionnalités d'une case sont assurées par deux types de *threads*, le *Thread Ecoute UDP* et des *Threads de Travail*. Chaque case possède un port de connexion TCP défini en mode programme. Afin d'établir une connexion TCP avec un pair, l'émetteur envoie par UDP le message «*Demande de connexion* », et attend la réception du message «*Connexion Acceptée* », pour préparer la connexion TCP de son coté.

III.2.1. Thread Ecoute UDP

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

III.2.2. Pool de Threads de Travail

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

III.2.3. Thread Ecoute TCP

Pour une meilleure gestion des connexions TCP/IP, économisant le temps de connexion entre deux pairs, une nouvelle architecture appelée *SDDS-TCP* à été rajouter au prototype SDDS2000. Cette nouvelle architecture permet d'une part l'ouverture passive[1] de connexion TCP/IP,qui apprête une case à accepter une connexion TCP/IP et réceptionner un tampon, et d'autre part la

[1] *Une ouverture passive signifie que le processus de connexion se met en attente d'une demande de connexion plutôt que de l'initier lui-même.[ISI81]*

gestion par l'acceptation de demandes de connexion TCP/IP concurrentes. Une fois que le *Thread Ecoute TCP* est lancé, au niveau de chaque case, il reste à l'affût des demandes de connexions entrantes. Il accepte et identifie le traitement à effectuer en analysant l'entête du tampon envoyé [M04]. Ce *thread* écoute sur le port *Port Ecoute TCP* réservé à la case, en utilisant le protocole TCP/IP.

III.3. Organisation d'une case LH*rs

Vu le grand volume de données qu'une case LH* doit gérer, une organisation de celle-ci interne s'impose. En effet, une solution a été proposée dans [KLR96], ce qui a donné une nouvelle variante de LH* dite LH*lh. Notons que SDDS2000 support LH*lh (section I.4.2). Une case de donnée LH*rs est organisé suivent LH*lh. La structure d'une case LH*lh est illustré par la ci-dessous.

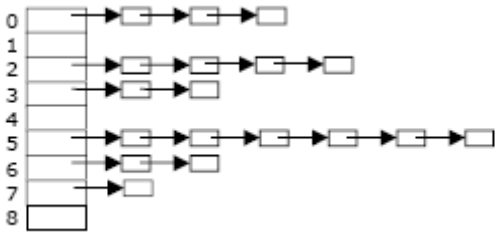


Figure 8. Structure d'une case LH*lh

Conclusion

Dans ce chapitre, nous avons survolé les principes de conception des SDDS et l'état de l'art dans la matière. Nous avons présenté tout particulièrement LH* et LH*rs. Notre intérêt pour ces SDDS est motivé par le fait que nous nous sommes basé sur elles pour concevoir notre système, présenté progressivement dans ce qui suit.

*LH*P2P : Hachage Linéaire Distribué et Scalable Pair à Pair*

Dans ce chapitre, nous présentons une généralisation du LH* aux environnements pair à pair. D'une manière générale, nous désignons la SDDS qui en résulte comme *LH*P2P*. Notre travail concerne néanmoins plus spécifiquement la conception et l'implémentation d'une variante de LH*P2P, offrant en plus la haute disponibilité scalable de LH*rs. Nous désignons cette variante par LH*rsP2P. L'aspect haute disponibilité n'a pas d'incidence sur les principes de généralisation que nous proposons. En d'autres termes, ils sont les mêmes pour LH*P2P et LH*rsP2P. L'architecture fonctionnelle et système du LH*rsP2P utilise néanmoins celle du prototype de LH*rs, [M04]. Ce prototype a des propriétés spécifiques que nous avons dû prendre en compte. Nous parlerons donc en général de LH*P2P. Nous discutons les caractéristiques spécifiques de notre implémentation de LH*rsP2P au fur et à mesure de la spécification du fonctionnement de notre SDDS.

Nous présentons d'abord l'idée spécifique dans LH*P2P par rapport à LH*. Il s'agit d'une conception spécifique de tout pair serveur d'un fichier LH*P2P, par rapport au pair LH*. Le gain majeur est la réduction du nombre maximal de renvois d'une requête de recherche d'adresse émise par un pair, à partir d'une clé d'un enregistrement. Dans LH* ce nombre est de deux renvois. En conséquence si le fichier LH*P2P ou LH*rsP2P, ne comporte que des noeuds pairs ou serveurs, toute requête de recherche d'un enregistrement a besoin au maximum d'un renvoi. Ceci fait de LH*rsP2P l'algorithme d'adressage P2P *le plus rapide dans ce sens à l'heure actuelle*. Pour rappel, la performance typique d'un algorithme d'adressage P2P utilisant une DHT est $O(\log N)$. L'adressage de LH*rsP2P est d'ailleurs optimal dans le cadre de principes d'une SDDS et du P2P en général. En effet, dans ce cadre un nœud client ou serveur de données typiquement ne connaît pas l'état exact du fichier. Il peut dès lors faire au moins une erreur d'adressage.

Après la présentation générale de LH*P2P, nous discutons les détails de l'algorithme.

I. Présentation générale de LH*P2P

LH*P2P fonctionne comme LH* aux extensions près suivantes. LH*rsP2P et LH*rs notamment fonctionnent d'une même manière en ce qui concerne la *k*-disponibilité.

Pour présenter les principes spécifiques de LH*P2P, rappelons d'abord que sous LH* un nœud pair comporte la partie serveur et la partie client, fonctionnant de la même manière qu'un serveur et un client quelconque de LH*. Il en est de même pour un pair de LH*rs. Sous LH*P2P, un pair comporte les mêmes composantes. Elle doivent toutefois gérer un même fichier et fonctionnent d'une manière spécifique. La Figure 9 et la Figure 10, illustrent les principes correspondants. Sous LH*rsP2P spécifiquement, on ne considère pour le concept du nœud pair que le serveur de données, pas celui de parité.

Egalement comme pour LH*, un pair LH*P2P gère en général une case de données. Si besoin on désigne un tel pair comme pair *serveur*. Il peut aussi être en attente d'assignation d'une case par le coordinateur. Nous l'appellerons alors pair *candidat* ou *en disponibilité* (ang. spare peer). La première différence majeure entre un pair LH* et celui de LH*P2P est que tout pair de LH*P2P ayant une case, émet un message IAM spécifique. Cet IAM dit de *rafraîchissement* est émis à tout éclatement de la case. Il part, d'une part au client du pair lui-même, client local. D'autre part, il peut partir vers certains pairs candidats.

L'IAM de rafraîchissement comporte le niveau *j* de la case sur le pair d'avant l'éclatement à faire. Le pair ayant reçu cette IAM met à jour son image (*i'*, *n'*) du fichier. Il suit l'algorithme (A) ci dessous. Ici *m* est l'adresse logique du pair émetteur ; *m* =0,1,…*N*-1. *N* est le nombre de nœuds.

(A)

```
i' = j  ;                          /* Image du niveau i du fichier
n ' = m  +1 ;                      /* Image du pointeur n d'éclatement
if n' = 2^i'  then i' = j + 1 ;  n' = 0 ;   /* Correction si le pointeur doit  revenir à zéro
```

L'image ainsi rafraîchie est exacte, au sens d'être celle (*i*, *n*) du fichier. Elle restera exacte, jusqu'au prochain éclatement du fichier quelque part. Tout calcul d'adresse d'un enregistrement par le client pendant ce temps ne peut donner lieu à aucune erreur d'adressage donc à aucun renvoi.

Par ailleurs, LH*rsP2P a un traitement spécifique d'un nouveau nœud qui se déclare comme pair au fichier. Comme tout nœud candidat qui veut être un client où serveur sous LH*, un tel noeud commence par contacter le coordinateur. Sous LH*, il peut se déclarer comme client ou serveur. Il devient pair s'il s'est déclaré une fois comme client et une autre fois comme serveur. Sous LH* P2P, il peut s'annoncer aussi d'emblée comme de type pair. Il deviendra alors typiquement le pair candidat. Il fonctionnera alors seulement comme client, ayant donc l'accès aux données. Il deviendra pair serveur uniquement quand un éclatement aura lieu et le coordinateur lui aura alloué la case *N* courante.

Les pairs candidats de LH* P2P sont assujettis à une gestion particulière. Le coordinateur assigne la gestion de tout candidat à un pair serveur ou à un serveur. Cette délégation est dite le *tutorat* du candidat. Le pair/serveur de données est dit *tuteur* et le candidat est son *pupille*. La fonction caractéristique d'un tuteur est le rafraîchissement de tout son pupille (en plus de son client local). Un tuteur peut avoir plusieurs pupilles, ou aucun. Afin de répartir uniformément la charge du tutorat sur tous les pair/serveurs de données du fichier, l'algorithme de l'assignation est dynamique. Il se base sur le hachage linéaire. Nous le présenterons dans ce qui suit. En

général, la moitié des pupilles d'un tuteur qui éclate va vers le nouveau nœud qui en devient le tuteur à son tour.

Le tuteur assure le tutorat d'habitude jusqu'à ce que le pupille l'avertisse qu'il n'est plus le candidat. En général, parce que il a reçu sa case. Mais, il peut aussi décider de quitter le fichier (quitter le réseau). Le tuteur peut aussi rayer un pupille de sa liste si le pupille n'acquitte pas d'IAM de rafraîchissement.

Entre temps, tout pupille fonctionne d'une part comme un client de LH* (ou LH*rs) et d'autre part comme un nœud LH* en disponibilité. Dès qu'il reçoit la case, son statut devient celui du pair serveur. Comme nous l'avons dit, il avertit alors son tuteur courant par un message approprié.

Le tutorat ne concerne que les pairs. En effet, une candidature pour être pair comporte implicitement un engagement de disponibilité nécessaire pour être un serveur de données. Un nœud client n'a pas une telle contrainte. En contrepartie, il serait impossible de rafraîchir son image. D'où notre politique générale de ne pas le faire. Un nœud client peut néanmoins demander à être serveur aussi, devenant alors un pair (candidat d'abord). De même, un nœud serveur de LH*rsP2P peut s'adjoindre la partie client, devenant pair à sont tour. Nous ne traitons pas ces cas particuliers davantage dans ce qui suit.

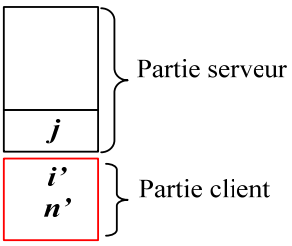


Figure 9. Architecture d'un nœud pair LH* P2P

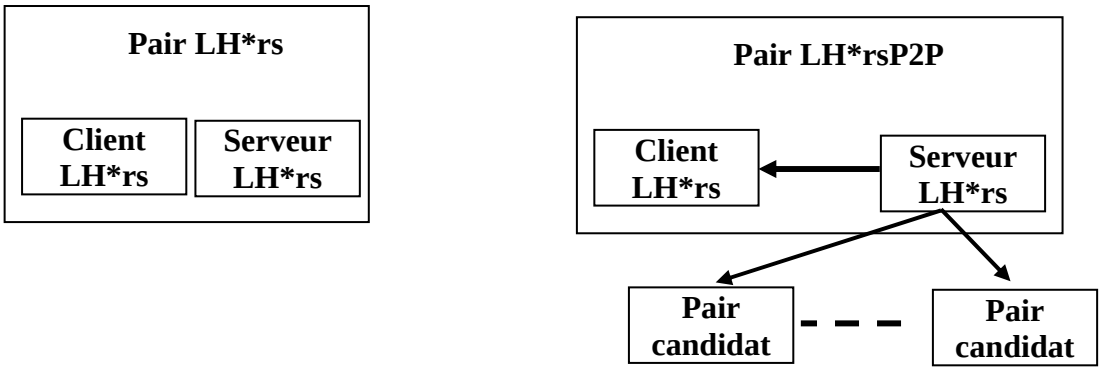


Figure 10. Architecture d'un nœud pair LH*rs et LH*rsP2P

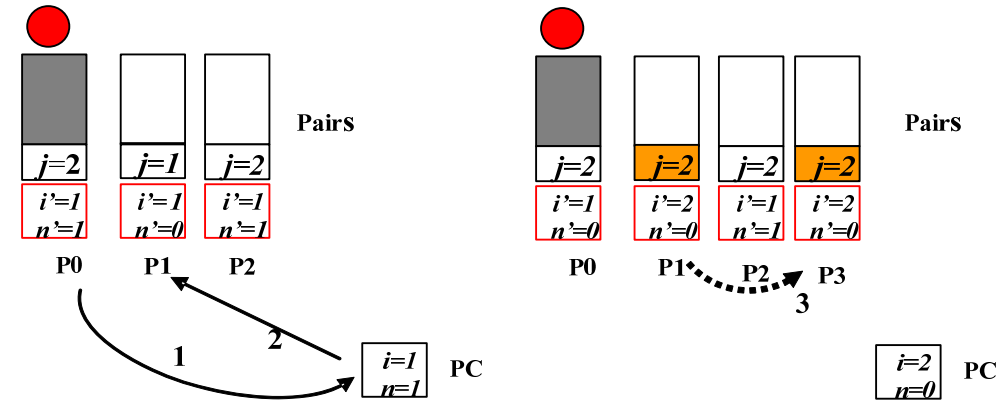
II. Eclatement d'une case sous LH*P2P

Soit *b* la capacité d'accueil d'un pair *P*. Dès qu'un pair est en débordement (dépassement de capacité), il informe le coordinateur. Ce dernier assigne la nouvelle case qui portera l'adresse logique (numéro) *N,* qui est en fait $N = n+2^i$, à un des pairs candidats. Il envoie alors la demande d'éclatement au nœud désigné par le pointeur d'éclatement *n*. Notons que, comme en général pour LH*, ce pair n'est pas nécessairement celui qui est en débordement. La demande comporte notamment *N* et l'adresse IP correspondant, ainsi que le numéro d'entité de *N* (attribué par le

coordinateur au nœud *N* spécifiquement pour LH*rs). Le pair pointé par *n* fait éclater sa case. C'est pareil pour LH* en général pour LH* P2P et la variante LH*rs P2P.

Suivent alors les opérations nouvelles spécifiques à LH* P2P. A savoir - d'abord, le pair *n*, met à jour l'image de sa composante client**,** à travers l**'**algorithme (A)**.** Puis, il envoie les IAM de rafraîchissement à tous ses pupilles. Ensuite, il détermine la liste des pupilles qui passent sous le tutorat du nouveau pair *N*. A cette fin, il hache les adresses IP des pupilles en utilisant la nouvelle valeur de *j*. Il envoie alors la liste au pair *N*. Ce pair l'enregistre et enfin, met à jour l'image *(i', n')* de sa propre composante client.

Nous illustrerons le mécanisme par un exemple dans ce qui suit, (cf. Figure 11). L'état du fichier est *(i, n) = (1, 1)*. Le fichier comporte initialement trois pairs,P0, P1,P3 qui comportent les cases 0,1,2 respectivement.



a) A vant éclatement **b) Après écl atement**

Figure 11. Eclatement d'un pair

1. Le pair P0 qui est le noeud de la case 0 informe le pair coordinateur qu'il est en débordement. En général et pour le prototype LH*rs que nous re-utilisons, c'est le nœud de la case 0 (nœud 0) lui-même qui héberge le coordinateur.
2. Le coordinateur fait le choix du pair serveur de la case 3. Les modalités de ce choix peuvent être diverses. Dans notre prototype, c'est le pair candidat qui répond le plus rapidement à la requête multicast du coordinateur qui devient le serveur de la case. Ce pair n'est pas montré sur la Figure 11. Notons que son tuteur ne pouvait être que P0 ou P1. Donc son image pouvait être (1,0) ou (1,1), mais pas (0,0).
3. Le coordinateur envoi un message d'éclatement au pair P1 (le noeud 1) désigné par le pointeur d'éclatement *n*. Une fois l'éclatement accompli, le coordinateur mettra à jour son image *(i,n)* à (2, 0).
4. L'éclatement du pair P1 est effectué, créant le pair P3 (nœud 3). A l'issue P1 met à jour son *j* à *j* =2 et son image (*i'*, *n'*) à (2, 0), selon l'algorithme *A*. Il n'a pas de pupille donc il transmet à P3 la liste vide.
5. Le pair P3 met à jour son image aussi à (2, 0), quel que soit sa valeur précédente.

Notons qu'après l'éclatement, les images de P1 et de P3 sont correctes. Celles de pairs P0 et P2 sont devenues incorrectes. Elles ne pourront être corrigées que par les futurs IAMs.

Enfin notons que pour LH*rs P2P spécifiquement, l'éclatement comporte aussi la mise à jour de cases de parité. Voir [M04] pour les détails.

III. Adressage

L'algorithme d'adressage de LH*P2P est celui de LH* côté client ou serveur. L'adressage d'une requête à clé d'une application commence sur le côté client d'un pair serveur ou de pair candidat. Il continue sur au moins un pair serveur ou serveur.

Toute composante client d'un pair utilise son image du fichier *(i',n')* et l'algorithme usuelle de LH* ci-dessous

a' ← hi'(C) ; /* a' est l'adresse du pair destiné à recevoir la clé C*/
if a' < n' then a ← $h_{i'-1}$(C) ; ……(A1)

La composante serveur qui reçoit les clés, vérifie si elles lui sont adressées et qu'il n'y a pas d'erreur d'adressage. En cas d'erreur la clé est redirigé vers un autre pair. Le tout selon l'algorithme de LH* que nous rappelons

a' ← hj(C) ; …….. ……..…….(A2)
if a'≠a then /* en cas d'erreur d'adressage*/
a" ← hj_{-1}(C) ; /* a" l'adresse destinée à recevoir la clé C */
if a"> a and a"<a' then a' ← a";

III.1. Ajustement de l'image du pair

En cas de renvoi, donc d'erreur d'adressage, la composante serveur du pair correct envoie au pair ayant émis la requête un IAM (Image Adjustement Message). Celui-ci a le même contenu que IAM de LH*. Le client exécute alors l'algorithme d'ajustement d'image de LH* qui est :

i'←j-1, n'← a+1 ; /* a est l'adresse du bon pair*/
if n' ≥ $2^{i'}$ then n'←0; i'← i'+1; ……..(A3)

IV. Insertion d'un nœud candidat

Un *nœud candidat* qui veut rejoindre le réseau, adresse sa demande à n'importe quel nœud du réseau. Le nœud recevant la demande contacte le coordinateur. L'image *(i', n')* du nouveau noeud est initialement à zéro. Le pair coordinateur décide du moment de l'insertion du nouveau nœud. Nous distinguons deux méthodes, la première est que le coordinateur décide d'éclater un pair prématurément, en d'autre terme avant que celui-ci ne déborde. Ainsi le noeud candidat est inséré dans l'immédiat, mais cette méthode ne garantit pas un taux de remplissage[1] que celui de LH* usuelle. La deuxième méthode est d'assigner un tuteur au candidat, donc le candidat devient un *pupille*. Dans le but de garantir un seul renvoi pour LH*P2P le nœud tuteur met à jour l'image du *nœud candidat* dont il a la charge (tous ses pupilles). A l'arrivée des noeuds candidats, le nœud coordinateur se charge d'attribuer à chaqu'un d'eux un tuteur. La distribution des *nœuds candidats* est uniforme, évitant ainsi d'avoir le cas où un seul *pair/serveur tuteur* prend en charge plusieurs *nœuds candidats*. La distribution est faite en hachant leur l'adresse, utilisant

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

l'image réel du fichier disponible au niveau du pair coordinateur. Toutefois nous signalons qu'il existe aussi une autre méthode pour insérer un nœud pair. En effet le pair candidat adresse sa demande au pair 0, qui héberge le coordinateur, ce dernier l'insère au moment opportun, en attendant il lui attribut un tuteur. C'est ce dernier mécanisme que nous avons utilisé pour la mise en œuvre de notre travail, expliqué dans le chapitre suivant.

IV.1. Avant insertion

Dans ce cas le noeud candidat se comporte comme un client LH*, son image du fichier *(i',n')* est initialisé à celle de son tuteur. À chaque éclatement d'un pair, celui-ci se charge d'envoyer sa nouvelle image au nœud dont il a la charge. Le mécanisme de mise à jour reste le même dans le cas d'une erreur d'adressage. La Figure 12.a montre ce processus.

IV.2. Après insertion

Le nœud coordinateur insère le nouveau noeud dès qu'un éclatement se produit. Ainsi on aura un nouveau pair qui a une partie serveur avec le niveau de la fonction de hachage *« j »*, une autre partie client avec l'image du fichier *(i',n')*. Il mettra à jour son image suivant l'algorithme **(A)** présenté précédemment. (cf. Figure 12).

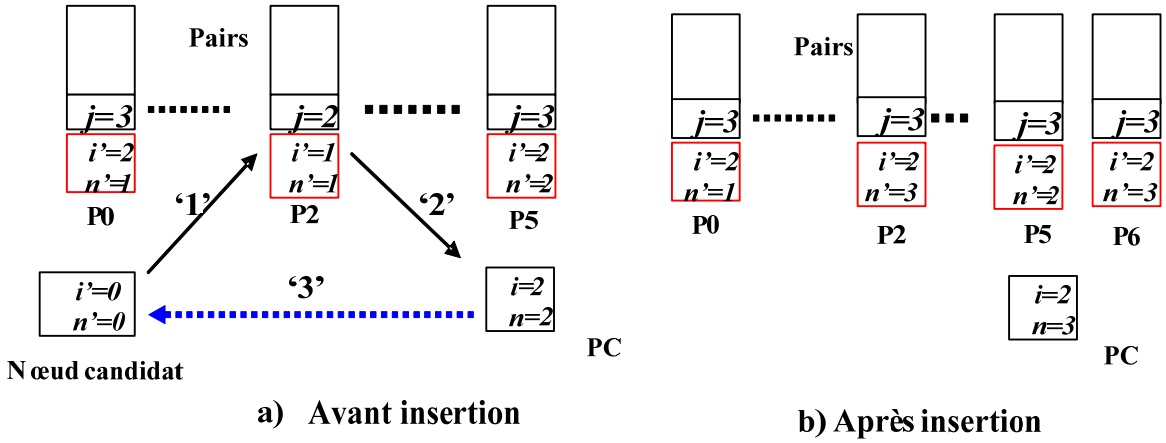


Figure 12. Insertion d'un nouveau pair dans LH* P2P

- '1' : le nouveau nœud contacte un pair quelconque (P2) en attendant son insertion, il peut tenir le rôle d'un client SDDS.
- '2' : le pair (P2) recevant la demande du nœud candidat informe le pair coordinateur de son arrivée pour qu'il puisse l'insérer. L'insertion du nœud candidat est faite lors d'un nouvel éclatement. Le nouveau nœud devient alors le pair P6.
- '3 ' : le pair coordinateur désigne un tuteur au noeud candidat. L'adresse du tuteur est le résultat du hachage de l'adresse IP (comme s'il s'agit d'une clé) du nœud candidat par coordinateur.
-

V. Expansion d'un fichier LH*P2P

Un fichier SDDS peut s'étendre, théoriquement, d'un à un nombre illimité de cases. Nous illustrons l'expansion d'un fichier LH*P2P par l'exemple suivent. Initialement le fichier est constitué d'une seule case de données, donc un seul pair. Les paramètres du système *(i,n,j,i', n')* sont initialisés à zéro, (cf. Figure 13.).

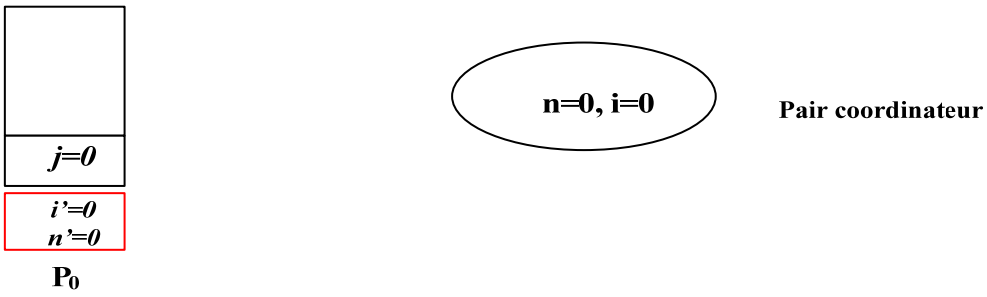


Figure 13Le fichier LH*P2P à l'état initial

Insertion des clés 1, 3, 4, 6 dans P_0. L'insertion de la clé 7 provoque un éclatement. L'adresse du pair est m=0, P0 met à jour son image avec l'algorithme A donc *i'=0, n'=1,* (cf. Figure 14).

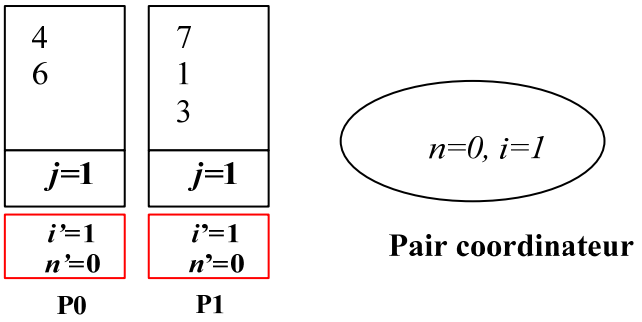


Figure 14. Expansion du fichier LH*P2P à deux cases

Insertion de 2,8. L'insertion de 10 provoque un éclatement. L'adresse du pair est *m*=0, mise à jour le l'image de P0, *i'=j*=1, *n'=m*+1=1 suivent l'algorithme A, (cf. Figure 15)

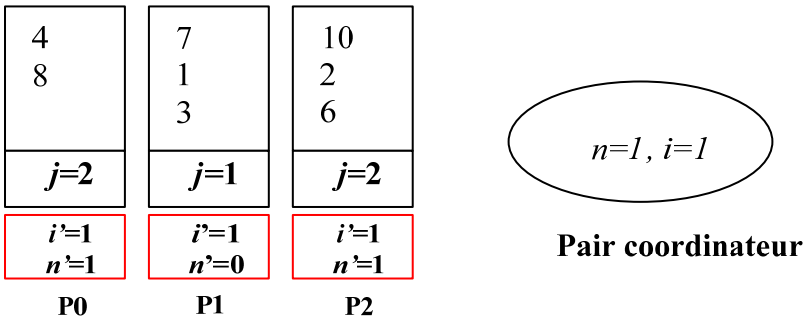


Figure 15. Fichier LH*P2P à trois cases

Insertion de 5, le 11 provoque l'éclatement. Le pair qui a éclaté est P1, ayant pour adresse *m*=1 d'où *i'*=2, *n'*=0. (cf. Figure 16*)*

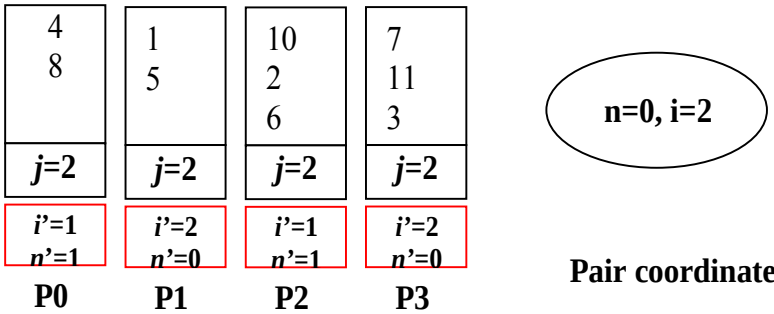


Figure 16. Fichier LH*P2P à quatre cases

Ainsi de suite le processus d'insertion. Au bout d'incertain nombre d'insertion on aura le fichier distribué sur les pairs comme le montre la figure ci-dessous. Le prochain pair à éclater sera P2 et ainsi de suit… (cf. Figure 17)

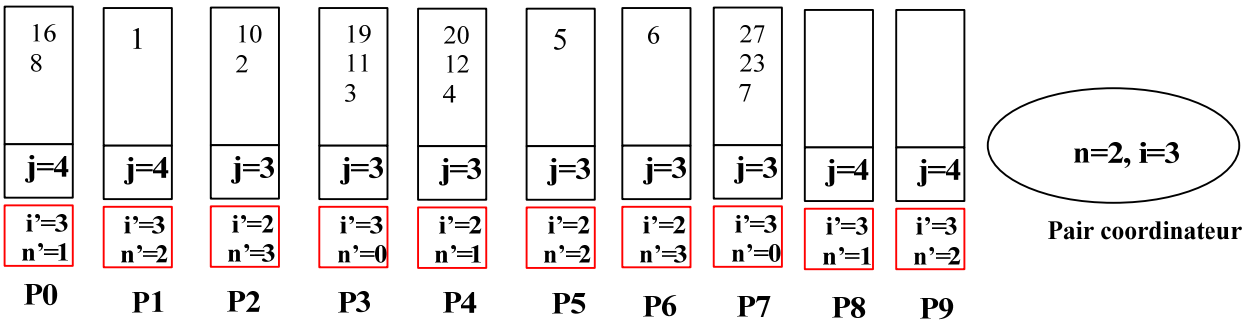


Figure 17. Fichier LH*P2P a dix cases

Lors de l'expansion du fichier LH*P2P, les pairs peuvent avoir une image incorrecte du fichier d'où l'erreur d'adressage. La Figure 18, ci-dessous montre ce processus.

Insertion avec une image fausse, le pair P_4 insère la clé 9 donc a'= $h_2(9)$=1. P1 exécute ***A2.*** Puis renvoi la clé 9 à P9, P9 envoi un IAM à P4.

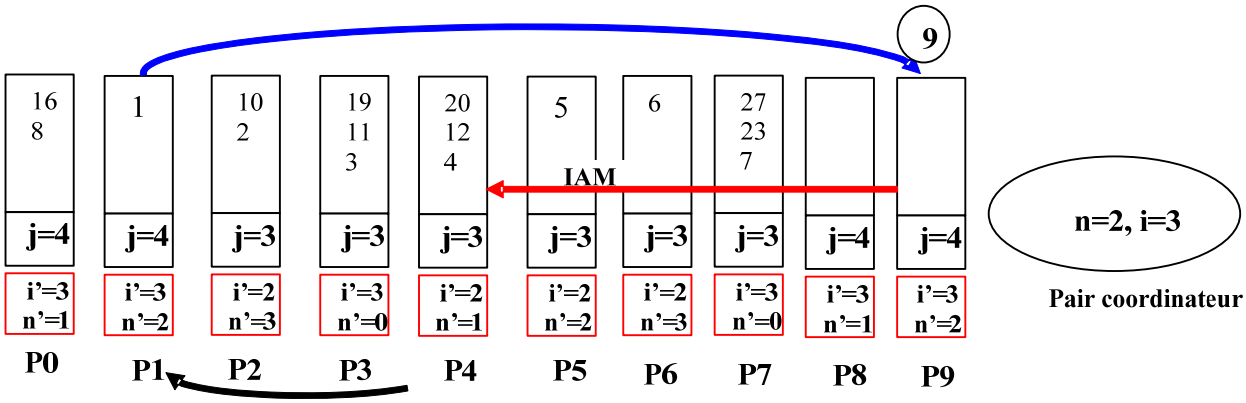


Figure 18. Insertion d'un article avec image incorrecte du pair

P4 exécute ***(A3)*** : i'←3, n'←9+1. On a $10 \geq 2^3$ alors n'←0, i'←3+1

Après mise à jour de l'image de **P4.** Si P4 recherche la clé 9 alors il aura la bonne adresse comme le montre la Figure 19 ci-dessous. P4 exécute ***(A1)*** : 9mod 2^4= 9, bonne adresse. (cf. Figure 19)

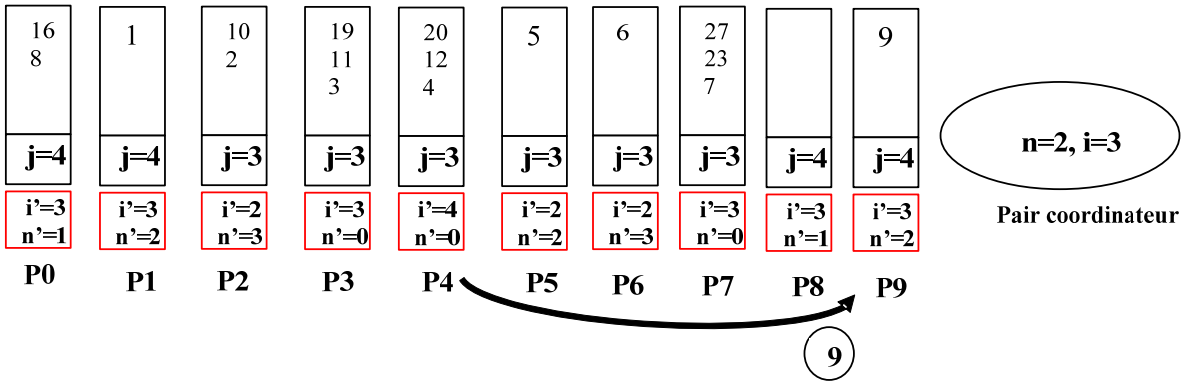


Figure 19. Mise à jour de l'image du pair, P4, et recherche de la clé 9

Conclusion

Dans ce chapitre, nous avons présenté l'algorithmique de LH* P2P. Ce dernier a la spécificité de réduire le nombre maximal de renvois dans le fichier de deux à un seul. Ce résultat ne peut être amélioré car le zéro de renvois implique une architecture centralisée ou inenvisageable en pratique. En effet l'algorithme A permet la mise à jour de l'image d'un pair lors d'un éclatement de sa case. D'où nous avons toujours deux pairs (celui qui a éclaté et celui issue de l'éclatement) avec l'image corrects du fichier et cela quelque soit la taille du fichier (le nombre de pair).

Notons que si le fichier LH*P2P comporterait également de clients LH* alors la propriété discutée ne serait valable que pour une requête émanant d'un pair. L'image du client ne pourrait pas être toujours rafraîchit, comme nous l'avons discuté. La borne de deux renvois de LH* d'une requête du client resterait en vigueur. Une telle version hybride de LH* P2P est envisageable, mais n'est pas notre objet.

Dans le chapitre suivent nous présenterons l'architecture fonctionnelle et système de notre implémentation de LH*P2P dans sa version LH*rsP2P. Cette description inclut notamment le traitement spécifique du « Churn » par LH*rsP2P, absent de la conception de LH* P2P.

Architecture Fonctionnelle du Système

Notre travail porte tout particulièrement sur la conception et l'implémentation de nouvelles fonctions spécifiques de LH* P2P en général et du LH*rsP2P en particulier. Nous allons maintenant présenter systématiquement nos solutions en ce qui concerne l'architecture fonctionnelle nous avons conçue. Ce travail nécessite la révision et l'extension d'un certain nombre de détails architecturaux de LH*rs dans sa version prototypée. Il s'agit aussi bien de la conception générale de l'algorithme que de certains aspects spécifiques au prototype. Nous discuterons ainsi de nouvelles tables système ainsi que celles modifiées. De même nous présenterons de nouveaux protocoles de messages, incluant de nouveaux types messages.

I. Adjonction d'un nouveau pair

Un nouveau noeud peut se déclarer comme candidat pour être le pair LH*rsP2P, ou seulement le serveur ou, enfin, seulement le client. Dans le cadre de nos travaux, nous nous intéressons au premier cas seulement. Les deux autres sont traités de la même manière que LH*rs, la Figure 20. présente le traitement fonctionnel d'une telle requête. Dans notre système, un pair candidat adresse sa demande, par un message noté *PairCandidat*, au coordinateur. Cette approche est la plus simple par sa compatibilité avec le protocole d'adjonction d'un client implémentée sous LH*rs. D'autres stratégies sont également possibles, où le candidat solliciterait un pair ou un serveur de sa connaissance. Le coordinateur commence par hacher l'adresse IP du candidat comme s'il s'agissait d'une clé d'un article. Le résultat sera l'adresse logique 1,2...*N* du pair ou serveur tuteur. Le coordinateur envoie ensuite un message, noté *NouveauCandidat,* au tuteur. Il y inclut les coordonnées du candidat. Le tuteur accuse la réception au coordinateur. Puis, il prend contact avec le candidat à son tour. Il lui envoie le message, noté *PriseEnCharge.* Le candidat accuse enfin la réception de ce message au tuteur. La structure des messages est décrite dans la section IV de ce chapitre.

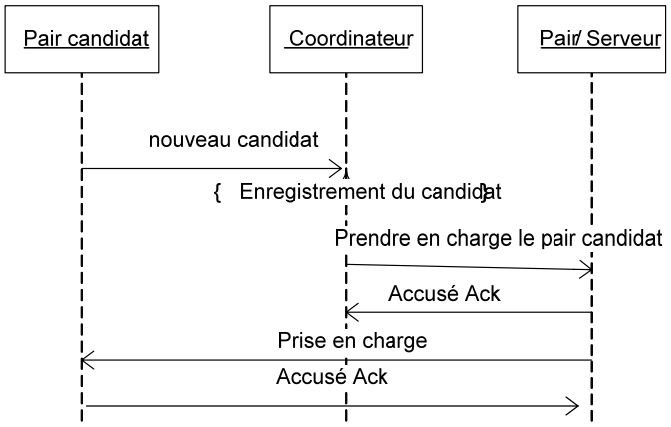


Figure 20. Protocole d'adjonction d'un pair candidat

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

II. Protocole d'éclatement

Les opérations de recherche, d'insertion, de mise à jour et de suppression d'article de LH*rsP2P sont les mêmes pour LH*rs, détails dans [M04]. Il n'est pas tout à fait de même pour l'opération d'éclatement qui peut suivre celle d'une insertion. LH*rsP2P comporte en effet une phase supplémentaire, (cf.Figure 21). Durant cette phase, il transmet au nœud avec la nouvelle case les adresses IP de pupilles qui seront désormais à sa charge. Celles-ci sont, pour rappel, toutes les adresses IP de candidats hachés vers la nouvelle case après l'éclatement. Le nouveau nœud prend désormais à sa charge le tutorat. Il envoie alors un message, *MiseAJourTuteur*, unicast à chacun de ces pupilles. Chaque pupille accuse la réception et met à jour son image.

Le nœud de l'éclatement rafraîchit par ailleurs chaque pupille qu'il avait sous son tutorat avant l'éclatement. Il utilise le message noté *MiseAJourTuteur,* (cf.Figure 21). Si c'est un

serveur seulement, alors c'est la fin de traitement d'éclatement spécifique LH*rsP2P pour lui. Autrement, il rafraîchit encore l'image interne de son client par le même message.

Les pupilles et peut-être le client local[1] ayant reçu le message, rafraîchissent leurs images. Les pupilles vérifient si leur tuteur n'a pas changé. Tout pupille peut le faire à l'évidence à partir de l'image. Dans la version actuelle du prototype, chaque pupille ou client local demande ensuite au coordinateur l'actualisation des adresses physiques (IP) de noeuds pairs/serveurs. Chaque pupille dispose le cas échéant dans cette liste reçue l'adresse IP du nouveau tuteur.

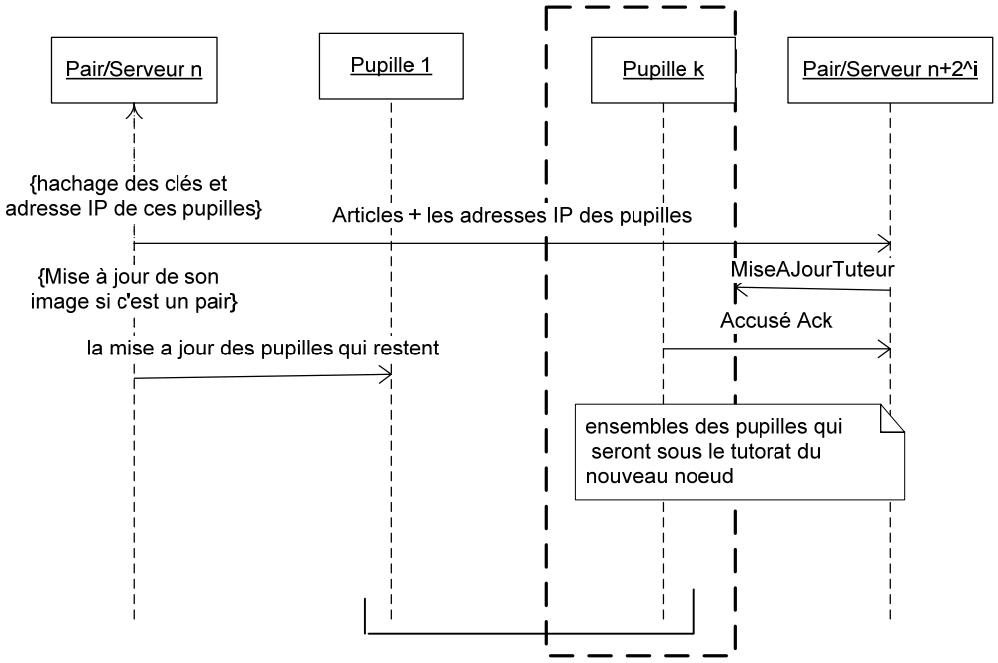


Figure 21. Protocole d'éclatement d'une case LH*rsP2P

III. Protocole de traitement du 'Churn'

Pour palier au 'churn' nous nous basons sur la gestion de *parité* de LH*rs. Les cases de données sont organisées en groupes de taille m ; $m = 2^l$, $l > 1$ étant en principe un paramètre de l'utilisateur. Chaque groupe est muni de $k \geq 1$ cases de parité, garantissant la k – disponibilité au groupe. A chaque reprise sur panne ou « churn » toutes les cases de parité sont mises à jour en ce qui concerne l'adresse de la case reconstruite. Si un pair quitte en effet le réseau sans avertir les autres noeuds, comme c'est typique du « churn », une requête à ce pair sera assimilée par LH*rs et par LH*rs P2P à celle d'une case en panne. La case sera reconstruite sur un autre noeud.

A la suite de la reconstruction, d'abord seul le pair ayant envoyé la requête et le coordinateur seront alors au courant de la nouvelle adresse. Un autre pair peut envoyer la requête au pair désormais absent. Soit k le numéro logique de la case concernée et donc du noeud. Après une attente le pair décrète alors que la case k n'est peut-être plus disponible. Sous LH*rs, il contacterait alors le coordinateur. Sous LH*rsP2P nous avons conçu une variante davantage scalable et potentiellement plus utile au « churn » (LH*rs était développé davantage pour l'environnement client/serveur ou grille). Le pair tente d'abord d'envoyer la requête, enrobée dans un message spécifique, aux autres membres du groupe, comme le montre la Figure 22. Le

[1] Un client local est un client LH*rs comme nous l'avons montré sur la Figure 10 de la page 20

calcul des adresses est trivial. Le premier membre qui reçoit ce message le transmet à une de cases de parité. Celle-ci transmet alors la requête au nœud correct. Ce dernier la traite, ou la renvoie. Le pair qui a initié la recherche reçoit le résultat et l'IAM correspondante. Il corrige alors son image.

Le pair à l'origine de la requête contacte le coordinateur, comme pour LH*rs seulement s'il n'y a aucune réponse de tout le groupe. Ceci est très peu probable, mais possible. Tous les membres d'un groupe de parité ont pu en effet migrer pendant que le pair ne s'adressait pas au groupe.

Il se peut aussi, dans le cadre du phénomène du « churn », qu'un pair quitte le fichier puis veut y revenir avec les données du fichier qu'il contient encore. Alors, il contacte d'abord le coordinateur. Celui-ci traite la demande en général comme celle d'un candidat. Les données du nœud devraient en effet, en général être déjà reconstruites ailleurs. Dans le cas rare où ce n'était pas encore le cas, le coordinateur autorise le pair à fonctionner comme auparavant.

Notons enfin le cas traité par LH*rs d'un serveur qui est coupé temporairement d'accès au reste du fichier sans le savoir et reconstruit ailleurs pendant la panne. Un client LH*rs peut alors s'adresser à tort à l'ancien serveur. Ce cas ne fait pas partie du « churn ». Nous le traitons donc dans LH*rs P2P comme dans LH*rs. Voir les détails dans [LMS04]

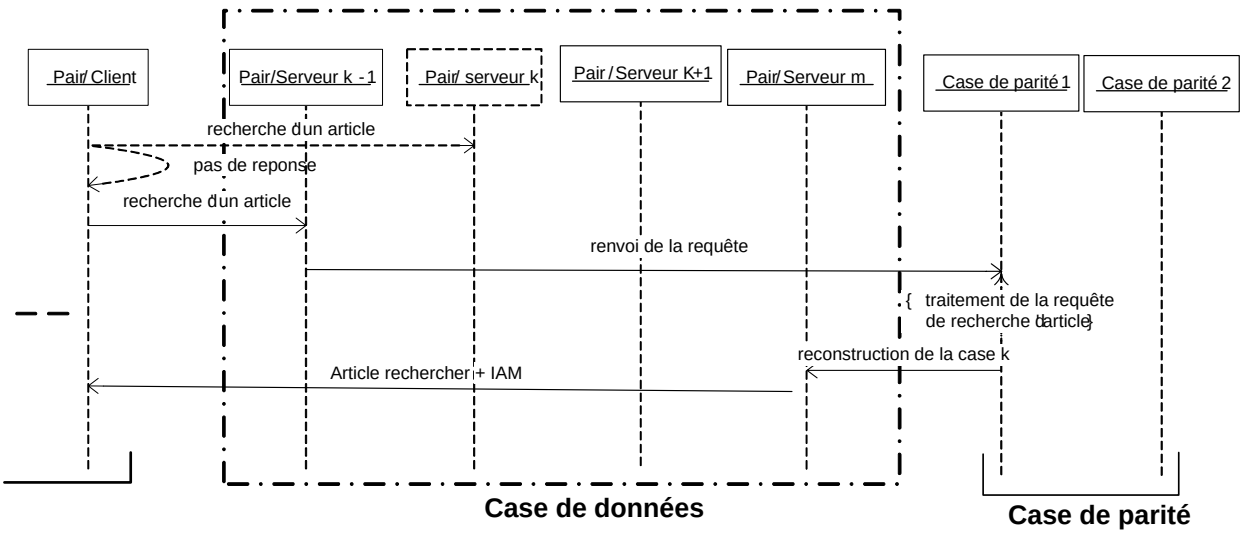


Figure 22. Traitement du 'Churn'

IV. Structures d'adressage de LH*rsP2P

Nous allons présenter les tableaux, que nous avons introduit pour pouvoir stocker et récupérer les paramètres nécessaires à l'adressage.

IV.1. Tables d'adressage du coordinateur

Comme dans LH*rs le coordinateur LH*rsP2P gère les éclatements et la haute disponibilité. Ainsi il se charge d'attribuer un *numéro d'entité* et un numéro logique à chaque nouveau nœud. Le numéro logique est attribué exclusivement aux cases de données [M04]. C'est le paramètre principale de l'adressage LH*.

Pour gérer les numéros d'entités, le coordinateur utilise le tableau 'Adresse des entités' de la Figure 23 noté 'TAE'. Le tableau est indexé par le numéro d'entité. Chaque élément du tableau

'TAE' a deux champs : *(Type Entité, Adresse IP)*. Le champ *Type Entité* indique pour chaque nœud connu du coordinateur si c'est un pair, un serveur de donnée ou un pair candidat. Le deuxième champ indique l'adresse IP du nœud.

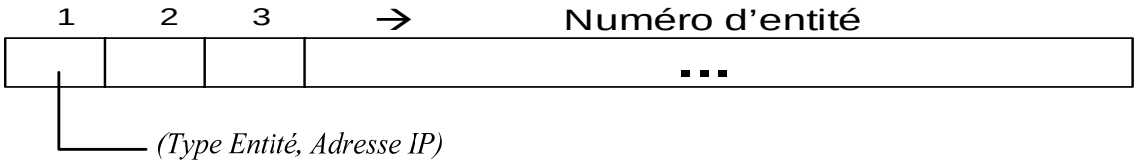


Figure 23. Adresse des entités (TAE)

Pour gérer les adresses logiques, le coordinateur utilise un deuxième tableau 'Numéros des cases de données' noté 'TCD' de la Figure 24. Ce tableau est indexé par un numéro logique. Un élément *i ; i*= 0,1,… contient le numéro d'entité de la case de données *i.*

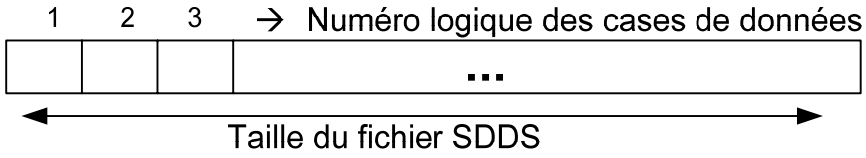


Figure 24. Numéros des cases de données (TCD)

Le coordinateur dispose aussi d'un tableau noté 'ATS' illustré dans la Figure 25 ci-dessous qui gère l'attribution d'un tuteur aux nouveaux sites. Celui-ci permet d'enregistrer les pairs candidats ainsi que leur tuteur. Un tuteur pourrait être un pair ou serveur. Chaque case du tableau ATS[*i*], contient une structure de tableau (T2), car un tuteur peut avoir zéro ou plusieurs pupilles. Chaque élément de T2 décrit un pupille par deux champs :

NuméroEntité : c'est le numéro d'entité du candidat, pour le calcul de son port.
AdresseIP : est l'adresse IP du candidat.

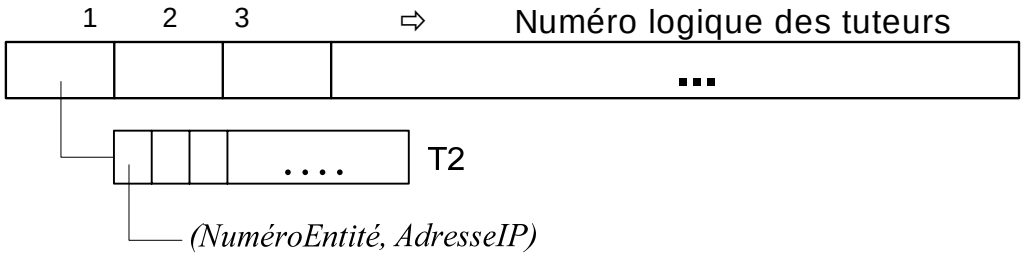


Figure 25. Attribution d'un tuteur aux nouveaux sites

Le tableau T2 de la Figure 25, est utilisé par chaque tuteur, afin d'enregistrer ces pupilles.

V. Structure des messages

Dans ce qui suit nous allons décrire la structure des nouveaux messages que nous avons crée pour les échangés entre les nœuds de LH*rsP2P. Nous discutons seulement les champs essentiels de chaque message correspondant spécifiquement à notre application.

V.1. Déclaration de candidature

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

PairCandidat(IDMessage, AdresseIP)
IDMessage : identifiant du message, qui est un numéro séquentiel unique sur deux octet. Tous les identifiants *IDMessage* sont sur deux octets.
AddresseIP : c'est l'adresse du nœud sur quatre octets (toutes les adresse IP sont sur quatre octets). La longueur du message est fixe.

V.1.1. Message du coordinateur au tuteur

Le coordinateur envoie au tuteur qu'il a choisi le message *NouveauCandidat.* La structure du message est comme suit :

NouveauCandidat (IDMessage, NuméroEntité, AdresseIP)

où :

IDMessage : identifiant du message.
NuméroEntité : numéro d'entité attribué par le coordinateur au candidat sur deux octets. Tous les numéros d'entités sont sur deux octets, de même pour les numéros logiques.
AdresseIP : adresse IP du pair candidat.

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

V.1.2. Message du tuteur au pair candidat

Une fois que le coordinateur a désigné un tuteur, ce dernier envoie un message, *PriseEnCharge*, au pair candidat lui indiquant son numéro d'entité et son adresse IP. Le message, *PriseEnCharge*, est de taille fixe, sa structure est la suivante :

PriseEnCharge(IDMessage, NiveauJ, NuméroLogique, NuméroEntité,AdresseIPTuteur)
IDMessage : identifiant du message.
NiveauJ : est le niveau' j' du tuteur, sur un octet.
NuméroLogique : numéro de la case sur le tuteur, (nécessaire pour l'algorithme A3 sur le candidat).
NuméroEntité : numéro d'entité du tuteur.
AderssIPTuteur : est l'adresse IP du tuteur.

V.2. Eclatement de la case d'un nœud

Comme nous l'avons déjà précisé dans la section III de ce chapitre, les messages destinés au traitement de l'éclatement sont comme suit :

Le message destiné à mettre à jours les pupilles et l'image du pair lors d'un éclatement est le suivant :

MiseAJourTuteur(IDMessage, NF_j, NuméroLogique, AdresseIPTuteur,NumeroEntité)
IDMessage : identifiant du message.
NF_j : niveau de la fonction de hachage '*j*'.
NuméroLogique : c'est le numéro de la case qui a éclaté, nécessaire pour l'exécution de A3.
AdresseIPTuteur : adresse IP du tuteur.
NuméroEntité : numéro d'entité du tuteur (qui va servir au calcul du port d'écoute sur le pupille).

Suit à l'éclatement, un pair candidat ayant un tuteur, peut être choisi pour devenir le nouveau pairt. A cet effet, il doit avertir son tuteur qu'il le quitte. La structure du message est la suivante :

FinTutorat(IDMessage, AdresseIP, NuméroEntité) où :
IDMessage : identifiant du message.
AdresseIP : adresse du nouveau pair qui a rejoint le réseau.
NuméroEntité : numéro d'entité qui va servir au calcul du port du pair.

Conclusion

Dans ce chapitre, nous avons présenté l'architecture fonctionnelle de notre système LH*rsP2P. Nous avons décrit les principales tables et messages correspondants. Ces tables et ces messages s'ajoutent aux celles et ceux déjà existants du prototype LH*rs, ou modifient l'existant par de nouvelles options ou paramètres. L'architecture décrite est à la base de notre prototype.

Domaine d'Application

Actuellement Internet est de plus en plus utilisé par les administrations publics et divers organismes. Dans l'optique de profiter des avantages d'Internet, les gouvernements européens membres de la CEE[1] peuvent interagirent et se rapprocher plus des citoyens. Ainsi les citoyens peuvent demander leurs papiers administratifs (acte de naissance, déclaration d'impôts…etc.) en ligne via un portail spécialement conçue pour ça.

Dans ce chapitre, nous présentons l'architecture générale du projet 'eGovernment', eGov. Nous montrerons la manière dont nous utiliserons notre SDDS LH*rsP2P, pour la gestion des documents du projet eGov.

[1] CEE : Communauté économique Européenne

I. Le projet 'e-Government', eGov

Le projet eGov a pour objectif de spécifier et de développer une plate-forme destinée à la mise en œuvre d'un *'portail administratif virtuel'*. Ce projet vise l'accès intégré aux services publics, nationaux et étrangers pour les citoyens et les entreprises de la CEE[1]. Ainsi, ces différents services seraient disponibles et accessibles en permanence. Les services serait catégorisés comme pertinents à un *événement de la vie* (ang, life event) ou un *événement d'entreprise* (ang, business event). Le projet se focalise sur les événements de la vie. Nous présentons maintenant plus en détail l'approche correspondante.

II. Les objectifs du "eGov"

Le projet "eGov" vise principalement à :

- La construction d'un "Portail administratif virtuel" unique et accessible via diverses technologies.
- La conception et la mise en œuvre d'un «Virtual Repository[2] », que nous notons 'VR'. Un 'VR' contient des documents 'eGov' et accessible via des 'Wrapper[3]' SQL, RDF..etc. En tant qu'un système de stockage, un 'VR' doit être scalable et distribué pour être capable de gérer une quantité de documents toujours croissante.
- La définition et le développement d'un "Governmental Mark-up Language", **GovML**, standard qui permet la description des documents "eGov". GovML est implémenté comme un dérivé de XML et fera office de 'colle' entre le portail et les « Virtual Repository » [G01]. la Figure 26, ci-dessous montre le vocabulaire 'GovML'.

[1] CEE : Communauté économique européenne.

[2] Virtual Repository : C'est un ensemble structuré de documents (***GovML***), constituant un cadre commun à plusieurs applications.

[3] Wrapper : programme enveloppant l'exécution d'un autre programme, pour lui préparer un environnement particulier utilisé par exemple pour sécuriser le fonctionnement de certaines applications en contrôlant très précisément leur interface avec le reste de l'univers commun.

	Public Services		Life Events
	Generic description	Specific description	Description
1	identifier	identifier	identifier
2	language	language	language
3	title	title	title
4	description	description	description
5	attention	attention	attention
6	faq-list	faq-list	faq-list
7	eligibility	eligibility	
8	required-documents	required-documents	
9	procedure	procedure	
10	periodicity	periodicity	
11	time-to deliver	time-to deliver	
12	cost-Info	cost-Info	
13	service-hours	service-hours	
14	employee-hints	employee-hints	
15	citizen-hints	citizen-hints	
16	related-services	public-authority-name	related-services
17	audience	Public authority department	
18	public-authority-type	e-documents	
19	law	delivery-channel	delivery-channel
20	result	cost	
21		contact-details	
22		service-code	
23		automation-level	
24		public-authority-address	
25		state	
26		service-name	

Figure 26. Vocabulaire GovML

Exemple : description d'un document eGov 'Life Event'. Le document est écrit en XML, les balises sont celles définies dans le vocabulaire de la Figure 26.

```
<?xml version=''1.0'' encoding='' UTF-8''?>
<govml:GovML xmlns:govml=''http://egov-projet.org/GovMLScheme/ ''
xmlns:xsi=''http://www.w3.org/2001/XMLSchema-instance''
xsi:schemaLocation=''http://egov-projetct.org/GovMLSchema/ file:///C:/temp/GovMLSchema.xsd>
        <description xsi:type=''govml:SpecificLifeEventDescription''>
                <identifier>ABC1234H</identifier>
                <language>EN </language>
                <title>Description of the life event''getting maried'' </title>
                <description> Getting married</description>
                <attention> This life event conccenes only adults </attention>
                <faq-list>
                    <item>
                        <question>Is there a possiblity toà get married online? </question>
                        <answer>Yes. Visiste the national governmental portal </answer>
                    </item>
                </faq-list>
                <related-services>
                    <item>
                        <title>Issuing a birth certificate </title>
                        <uri>http://www.egovproject.org\birth#</uri>
                    < /item>
                </related-services>
                <law>Law withe nhimber FRC-234</law>
        </description>
</govml:GovML>
```

III. Architecture du système 'eGov

L'architecture du système sur laquelle nous appliquerons LH*rsP2P est constituée de divers composants comme le montre la Figure 27, ci-dessous :

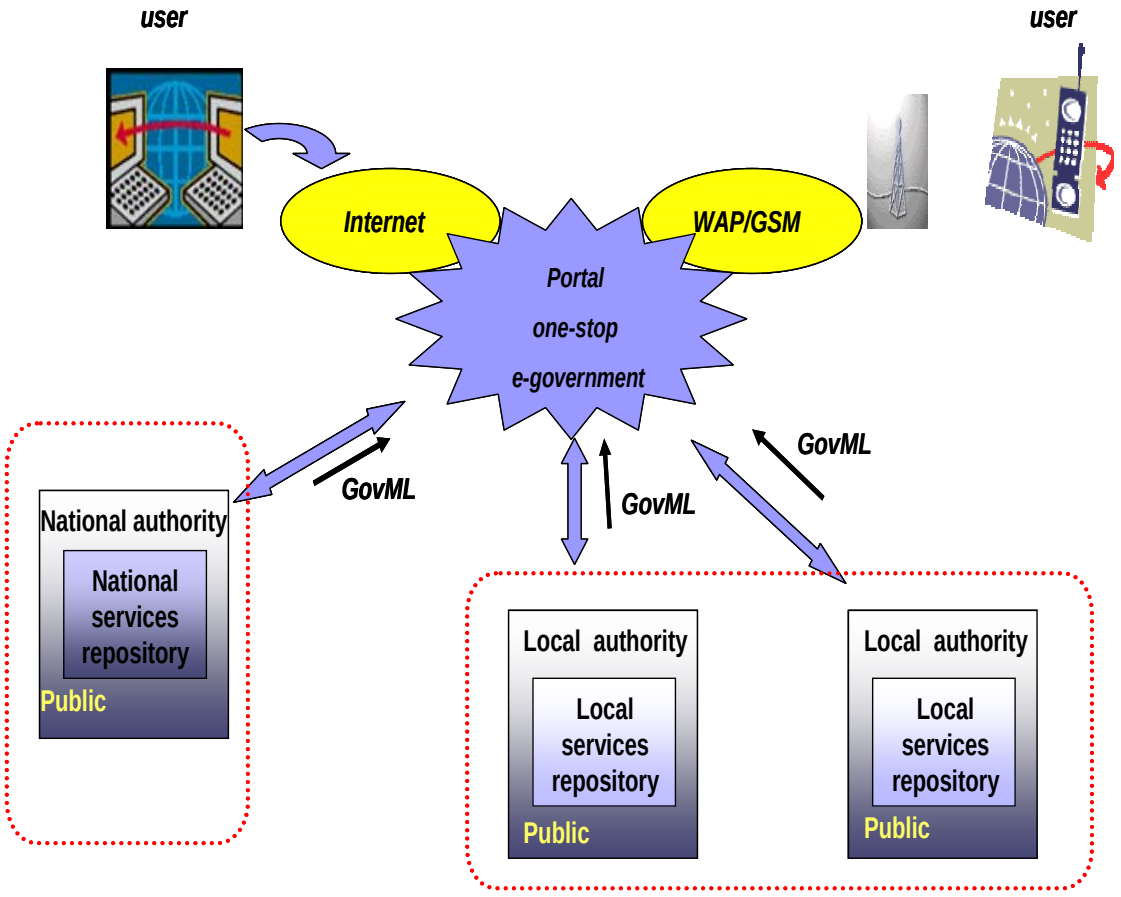


Figure 27. Architecture du système 'eGov'

Les principaux composant du système 'eGov' sont :

- Un portail unique, accessible par diverses technologies. C'est le point où les utilisateurs (citoyens, entreprises, et autorités publics) pourront accéder aux documents 'eGov' dont 'Life events' et 'Busness events'.
- 'GovML', le langage standardisé qui permet la définition et la description des documents 'eGov 'comme nous l'avons montré dans la section précédente.
- Les « VR » qui stockent les documents 'eGov' dont nous avons la charge de fournir une bonne architecture et outils pour gérer les « Virtual Repository ». C'est ce que nous allons présenter dans la section suivante.

IV. Architecture du « Virtual Repository »

L'objectif est d'applique notre SDDS pour gérer, une partie d'un 'VR'. Nous voulons assurer la scalabilité du système ainsi qu'une une haute disponibilité des données. Pour cela une architecture est proposée dans [LMS06] comme nous le montre la Figure 28.

La proposition consiste en l'utilisation de deux prototypes développés au CERIA[1]. Il s'agit de SDDS2005 [LMS03], qui est un système de gestion de fichiers distribués et scalables et SD-SQL Server [LRS02], un système de gestion de base de données distribué et scalable.

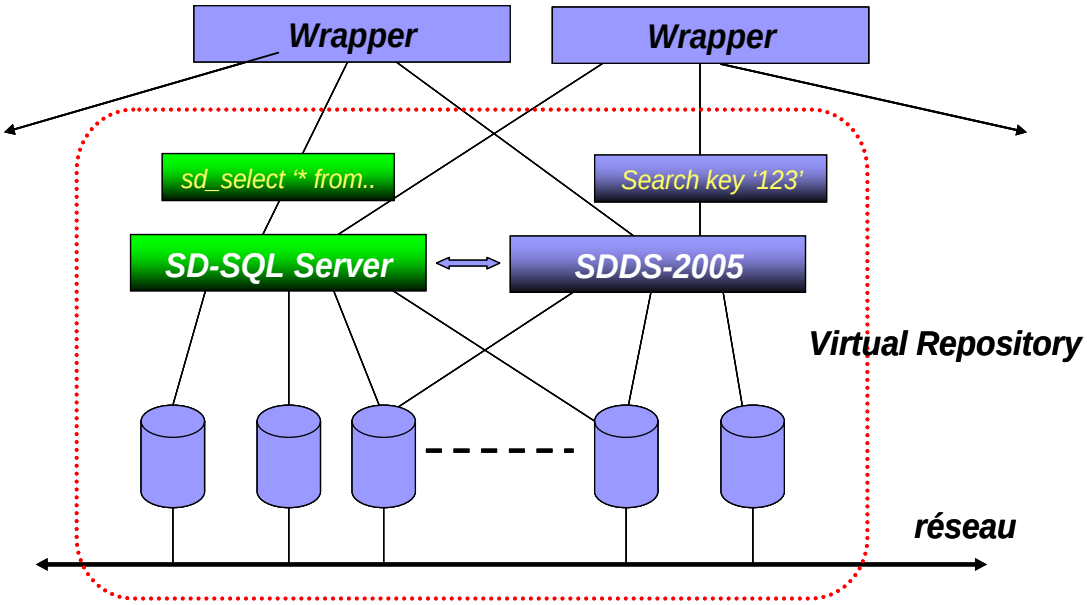


Figure 28. Architecture d'un «Virtual Repository »

Dans le but de garantir la haute disponibilité des données, nous remplaçons le prototype SDDS2005 par notre système. Ainsi un document *GovML* est représenter comme un article identifié par une clé sous notre système LH*rsP2P.

Le rôle de LH*rsP2P est alors, de géré les documents 'eGov' souvent utilisés et peu volumineux, tels que les formulaires vide (Acte de naissance, déclaration d'impôt…etc). Chaque document pourrait être stocké en tant qu'un document 'GovML'. SD-SQL Server est destiné à gérer les documents 'eGov' volumineux et peu utilisés. Les formulaires, une fois remplit sont pris en charge par SD-SQL Server pour le stockage. Notons que chaque un de ces prototype a ces propre commandes pour stoker et interroger les données stockées.

Conclusion

En somme, notre architecture sur la Figure 28 permet l'accès rapide aux documents 'eGov' et le stockage théoriquement sans limite de documents sous 'GovML' dans un VR. Notre composante propre, LH*rs P2P, pourrais assurer notamment la disponibilité de formulaires fréquemment utilisés dans le cas d'une panne ou départ d'un nœud d'un VR.

[1] CERIA : Centre de Recherche en Informatique Appliquée

Conclusion et perspectives

Nous avons proposé une nouvelle structure de données distribuées et scalable, LH*rsP2P qui utilise les principes du hachage linéaire. Cette nouvelle SDDS permet de construire des fichiers distribués sur des nœuds (sites) pairs et serveurs de données. Sa caractéristique principale est un renvoi au plus d'une requête à clé entre les pairs ou serveurs. Ce résultat est dû à la mise à jour de l'image du fichier de pupilles au moment de l'éclatement. Ce nombre de renvois est à notre connaissance unique actuellement à notre SDDS. LH* offre la performance de deux renvois au plus. D'autres organisations P2P connues telles que Chord par exemple nécessitent O (log N) renvois.

LH*rsP2P, offre notamment un mécanisme de récupération de données, compte tenu du problème du 'Churn'. Il s'agit d'une propriété connue des systèmes P2P où un pair ou plusieurs pairs peuvent aisément quitter le réseau sans en avertir les autres pairs ou tomber en panne, comme d'habitude. Notre mécanisme est une adaptation du celui de LH*rs à efficacité prouvée [LM02]. Nous protégeons contre le départ inopiné d'un nombre arbitraire de k pairs par groupe de pairs. La valeur de k et la taille du groupe sont des paramètres d'utilisateur, ou automatiquement ajustables selon la nombre de pairs.

LH*rsP2P devrait avoir de nombreuses applications. Notre intérêt particulier porte sur la gestion d'une partie d'un 'Virtual Répository' du projet 'eGov'. Nous pensons que notre système doit permettre la gestion des documents 'GovML' peu volumineux et souvent utilisés, surtout des formulaires vides relatifs aux divers événements de la vie 'Life envents'.

Notre travail ouvre plusieurs axes de recherche. Il s'agit dans l'avenir proche, pour nous, surtout des objectifs suivants :

- Construction d'une preuve formelle, afin de prouver que nous garantissons un et un seul renvoi dans LH*rsP2P et cela quelque soit la taille du fichier SDDS.
- Une étude expérimentale générale de notre implémentation de LH*rs P2P
- Application aux documents réels GovML du projet 'eGov'
- Etude dans ce cadre de variantes de LH*rs P2P que nous avons signalé.

Bibliographie

[AWD01] ***Ailamaki,*** *A. & al. DBMSs on a modern processor: Where does time go. In Proceedings of the 24th VLDB Conference, 1999.*

[Ben00] ***Bennour F.S*** *Contribution à la Gestion de Structures de Données Distribuées et Scalables, Thèse de doctorat, Juin 2000, Université Paris Dauphine*

[Bur83] ***W.A. Burkhard -*** *Interpolation-based Index Maintenance. Proc. of the 2nd Symp. On Principles of databases Systems. TODS 1983. pp.76-88.*

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

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

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

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

[G01] ***Glassey O ,*** *EPFL. Isps.ch newsletter n)11.10/2001*

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

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

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

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

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

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

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

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

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

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

[LMS04] ***Litwin, W, Moussa R, Schwarz T:*** *LH*RS: A Highly Available Distributed Data Storage Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004*

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

[LMS06] ***Litwin W, Mokadem R, Sahri R.*** *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] High-Performance Multi-Attribute Access. Res. Rep. GERM Paris 9& Distributed Inf. Techn. Dep. HPL Palo Alto, April 1995.

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

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

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

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

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

[LRS02] ***Litwin, W. & Sahri, S.*** *Implementing SD-SQL Server: a Scalable Distributed Database System. Intl. Workshop on Distributed Data and Structures, WDAS 2004,*

Lausanne, Carleton Scientific (publ.).

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

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

[M04] ***R Moussa.*** *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*

[Stoica01] ***Stoica, Morris, Karger, Kaashoek, Balakrishman,*** *Chord : A Scalable Peer to Peer Lookup Service for Internet Appication, SIGCOMM'0, August 27-31, 2001, San Diego, California USA*