Publication avant LAFMC

Titre	Implémentation d'une bibliothèque de primitives d'accès aux fichiers SDDS- RP*
Auteurs	Mohamed T. Seck, **Samba Ndiaye**, Witold Litwin, GerardLEVY, Aly W DIENE
Référence	4e Colloque africain sur la recherche en informatique et mathématiques appliquées (CARI 1998)
Editeur	CARI
Pages	481 - 493
Année	1998
DOI	
URL	http://horizon.documentation.ird.fr/exldoc/pleins_textes/pleins_textes_6/colloques2/010008708.pdf
Index	
ISBN	
Encadreur	Non
Extrait d'une thèse	Non

Implémentation d'une bibliothèque de primitives d'accès aux fichiers distribués et scalables RP*

Mouhamed T. SECK[1] , Samba NDIAYE[2], Witold LITWIN[3] Gérard LEVY[2], Aly W. DIENE[2]

Résumé :
Les fichiers distribués *scalables* ouvrent de nouvelles perspectives pour les applications développées sur les réseaux d'ordinateurs ou *multi-ordinateurs* [LITWI93], [LITWI93'], [LITWI95a], [LITWI96a]. Une des familles de SDDS, dite RP* qui préserve l'ordre, a été introduite dans [LITWI94]. Ces fichiers, résidant en mémoire RAM, offrent des performances d'accès aux données bien supérieures aux solutions classiques. Après avoir proposé une organisation interne pour les cases des fichiers RP*, nous présentons des structures et des algorithmes permettant d'implémenter les primitives de recherche, d'insertion, de requêtes par intervalles, etc...mettant ainsi en place une bibliothèque d'accès pour ce type de fichiers distribués.

Mots clé : fichier scalable et distribué, B-arbre, fichier mappé, multi-ordinateur, multicast, unicast, thread

Abstract :
The distributed and scalable files offer new opportunities for the software developped for the multi-computer architecture [LITWI93], [LITWI93'], [LITWI95a], [LITWI96a]. One branch of the SDDS family called RP* based on B+ tree structure has been introduced in [LITWI94]. These memory files are more efficient than traditional ones. We have implemented an access library for RP* files. First, we propose internal data structures of the file's buckets, and then we describe various algorithms to implement the basic routines such as search, insert, range query etc.... for RP* files.

1 Motivation

Les multi-ordinateurs

Nous sommes aujourd'hui à l'ère des réseaux informatiques. L'avènement d'Internet en est l'exemple le plus frappant. Certaines organisations ont mis en place des réseaux pouvant comporter jusqu'à des milliers d'ordinateurs, connectés par des câbles dont la bande passante peut atteindre le gigabit/s. De telles configurations sont à la base du concept de *multi-ordinateurs* [TANEN95]. Nous sommes en présence d'une évolution majeure de l'informatique. Les multi-ordinateurs constituent des ressources gigantesques, jusqu'ici inconnues, en terme de puissance de calcul processeur, de capacité de stockage disque et de capacité de mémoire centrale. Comment utiliser ces ressources, particulièrement la mémoire centrale ainsi disponible? Plusieurs équipes de recherche dans le monde y travaillent.
Une direction de recherche concerne la création d'un système de gestion de fichiers distribués dans les mémoires vives des ordinateurs du réseau. Les performances attendues pour ces fichiers s'expliquent par le fait qu'il est plus rapide d'accéder à une donnée en RAM distante qu'en mémoire secondaire locale sur les réseaux de type Ethernet [GRAY93]. Ces fichiers sont appelés *Structures de Données Distribuées et Scalables (SDDS).*

Les SDDS

Un fichier SDDS est constitué de partitions, appelées *cases*, installées dans les RAM de plusieurs ordinateurs différents. Une exigence importante d'un fichier SDDS est de pouvoir s'étendre dynamiquement sur le multi-ordinateur, et donc d'avoir une très grande taille.
Les machines qui stockent des partitions d'un fichier SDDS sont appelées *serveurs*. Dans cette étude on suppose qu'un serveur ne peut contenir qu'une seule case d'un fichier SDDS. Les autres machines sont dites *clients*. Ces notions de clients et serveurs ne sont pas exclusives, une machine pouvant être à la fois client et serveur. Les clients accèdent aux données d'un fichier SDDS en s'appuyant sur une

[1] Département Génie Informatique Ecole Supérieure Polytechnique (UCAD) seckm@ucad.sn.
[2] Département Mathématique et Informatique Faculté des Sciences et Techniques (UCAD) ndiayesa@ucad.sn. aldiene@ucad.sn, glevy@ucad.sn.
[3] Université Paris 9 Dauphine GERM/CERIA

"connaissance" du fichier. Cette connaissance est représentée par une structure de données, appelée *image* du fichier, qui permet à tout instant de savoir dans quelle case est supposée se trouver une donnée quelconque du fichier. Si la donnée recherchée n'est pas dans la case supposée alors un mécanisme de *redirection* permet à la fois de retrouver la bonne case et en retour *d'ajuster* l'image du fichier au niveau du client (*Image Adjustment Message ou* IAM.).
Pour éviter les goulets d'étranglement , il n'y a pas de répertoire central d'accès à un fichier SDDS.
Plusieurs SDDS ont été proposées LH*, RP*, etc... [LITWI93a], [LITWI94], [LITWI95a], [LITWI95b], [LITWI96a], [LITWI96b]. Les SDDS LH* sont toutes basées sur le hachage [LITWI93]. Leur inconvénient majeur est qu'elles ne préservent pas l'ordre. C'est pourquoi il est proposé une deuxième famille de SDDS dite RP* [LITWI94] qui préservent l'ordre des enregistrements; il s'agit de RP*$_N$, RP*$_C$ et RP*$_S$. Ces acronymes font référence à des variantes qui se distinguent par le protocole de communication entre client et serveurs. Notre travail consiste à proposer une implémentation des trois variantes de ce système de fichiers sur la plate-forme Windows NT et de valider nos hypothèses sur les gains de performance. Plus précisément, nous avons mis en place une bibliothèque de primitives de gestion des fichiers SDDS RP* (*insertion, modification, suppression, recherche, éclatement*);
La suite de cet article est organisée comme suit : la section 2 présente les SDDS RP* et met en évidence les caractéristiques des variantes RP*$_N$, RP*$_C$ et RP*$_S$. La section 3 est consacrée à la description de l'organisation interne de l'espace de stockage d'une case SDDS. La section 4 décrit l'algorithme d'éclatement d'une case. La section 5 spécifie la structure de l'image d'un client et de l'index du fichier reparti sur des serveurs spécialisés appelés serveurs de nœuds. La section 6 présente l'architecture du prototype qui a été implémenté et les résultats expérimentaux.

2 LES SDDS-RP*

Une SDDS bien conçue doit produire peu d'erreurs et peu de redirections de requêtes entre serveurs et doit permettre l'extensibilité de la performance d'accès lorsque le fichier devient volumineux.
Une étude théorique de la performance d'accès, basée sur le nombre de messages échangés entre clients et serveurs, est présentée dans [LITWI94].
La première variante RP*$_N$ définit un partitionnement des enregistrements par intervalles définis sur l'espace des clés. Le protocole de communication s'appuie sur des messages multicast[1] émis par les clients et des messages unicast pour la réponse des serveurs.
La seconde variante RP*$_C$ ajoute à RP*$_N$ une image au niveau de chaque client. Une telle organisation favorise l'utilisation des messages unicast[2] aussi bien lors des requêtes clients que des réponses serveurs; le multicast n'est employé que pour les redirections.
Enfin, RP*$_S$ ajoute un index reparti sur des serveurs de nœuds, ce qui permet l'utilisation de messages unicast lors des redirections.

RP*$_N$	Pas d'image, tout multicast
RP*$_C$	+ image sur le client, multicast limité
RP*$_S$	+ index sur les serveurs de nœuds, multicast optionnel

Fig.1 : Les trois variantes de RP*

L'implémentation de la totalité du système de fichier a été réparti en deux modules:
- Le protocole de communication entre clients et serveurs , avec la prise en compte des différentes variantes ;
- L'organisation interne du fichier SDDS ainsi que les primitives de gestion des données qui fait l'objet du présent article.

[1] Multicast : destiné à un ensemble de machines qui appartiennent à un même groupe sur un réseau donné.

[2] Unicast : destiné à une seule machine,

2.1 SDDS RP*$_N$

Structure et évolution du fichier

Les enregistrements sont stockés dans des cases de capacité b ($b >> 1$), ils sont logiquement ordonnés suivant les valeurs de clés croissantes ou décroissantes. Chaque case B est munie d'un en-tête contenant deux valeurs λ et Λ respectivement appelées *clé minimale* et *clé maximale*. Un enregistrement de clé c appartenant à B est tel que :

$\lambda < c \leq \Lambda$. L'intervalle $(\lambda, \Lambda]$ est appelé *intervalle* de la case.

Un fichier RP*$_N$ est initialement constitué d'une case unique (case *0*) d'intervalle $(-\infty, +\infty]$. Toutes les insertions initiales vont à la case *0*. Elle est éclatée quand elle déborde, cela entraîne la création d'une nouvelle case appelée *case 1*, et ainsi de suite. On suppose que la liste des serveurs potentiels est connue. Pour retrouver un enregistrement, un client envoie un message à tous les serveurs potentiels (message multicast). Chaque serveur ayant reçu le message vérifie si la clé de l'enregistrement demandé appartient à son intervalle $(\lambda, \Lambda]$. Si oui, il renvoie l'enregistrement demandé au client. Si non il ne fait rien.

2.2 SDDS RP*c

Dans RP*$_C$, on améliore le protocole en réduisant le nombre de messages en installant une *image du fichier* au niveau du client. Cette image donne l'adresse du serveur sensé contenir l'enregistrement recherché. Le client adresse alors un message directement au serveur concerné (message unicast). Si celui-ci ne contient pas l'enregistrement, il redirige la requête en envoyant un message multicast à tous les autres serveurs. Finalement le serveur qui possède l'enregistrement le renvoie par message unicast au client ainsi qu'un IAM.

2.2.1 Structure et évolution de l'image du client

L'image du fichier SDDS au niveau d'un client se présente comme une liste ordonnée de couples $T[i] = (a_i, C_i)$ où :

- a_i est l'adresse d'une case du fichier SDDS (les adresses inconnues sont notés *), en pratique c'est le numéro IP du serveur qui héberge la case.
- C_i est la *clé maximale* $\Lambda(i)$, elle représente aussi la *clé minimale* $\lambda(i+1)$ de la case d'adresse $i+1$
- Initialement $T = (0, \infty]$, elle évolue en fonction des IAM

L'algorithme de recherche d'un enregistrement de clé c est le suivant :

- Le client recherche dans son image le premier couple (a_i, C_i) tel que $C_i \geq c$. Si $a_i \neq *$ alors envoyer une requête unicast au serveur d'adresse a_i, si non envoyer une requête multicast à tous les serveurs.
- Un serveur qui reçoit une requête envoyée par unicast avec une clé c qui n'appartient pas à son intervalle, ajoute son intervalle et son adresse à la requête et la redirige par multicast vers les autres serveurs. Dans le cas contraire, il répond en donnant son intervalle, son adresse et l'enregistrement trouvé.
- Un serveur qui reçoit une requête envoyée par multicast avec une clé c qui n'appartient pas à son intervalle ne fait rien. Dans le cas contraire, deux situations se présentent :
  - Si la requête est envoyée par multicast à partir d'un autre serveur d'adresse k il répond en donnant son adresse, son intervalle, l'enregistrement trouvé, k et l'intervalle de la case d'adresse k.
  - Si la requête multicast vient d'un client, notamment dans le cadre d'une requête générale, il répond en donnant son adresse, son intervalle et l'enregistrement trouvé.

2.2.2 Ajustement de l'image du client

L'ajustement de l'image du client consiste à traiter les IAMs (λ, a, Λ) reçus où λ et Λ représentent l'intervalle du serveur d'adresse a. Pour chaque IAM :

(a) S'il n'existe pas de couples (a, C_a) appartenant à l'image T avec $C_a = \lambda$ et $\lambda \neq -\infty$ alors insérer $(*, \lambda)$ dans T.

(b) S'il existe un couple (a, C_a) appartenant à T avec $C_a > \Lambda$ alors : si $C_a = +\infty$ alors $C_a = \Lambda$ et ajouter $(*, +\infty)$ dans T

si non si $C_a < +\infty$ alors $C_a = A$.

(c) S'il existe un couple *(*, A)* appartenant à T alors remplacer * par a.

(d) S'il n'existe pas de couple *(a, A)* appartenant à T alors insérer *(a, A)* dans T.

La figure ci-dessous montre l'évolution d'un fichier SDDS RP*. Il est initialement constitué d'une case unique, vide. Les insertions initiales se font dans cette case. Elle est éclatée quand elle déborde et cela entraîne la création d'une nouvelle case qui comporte la moitié des enregistrements. L'intervalle *(– ∞ , + ∞]* éclate en deux intervalles *(– ∞ ,of] et (of , + ∞]*. Le fichier évolue ainsi d'un serveur à plusieurs serveurs de stockage.

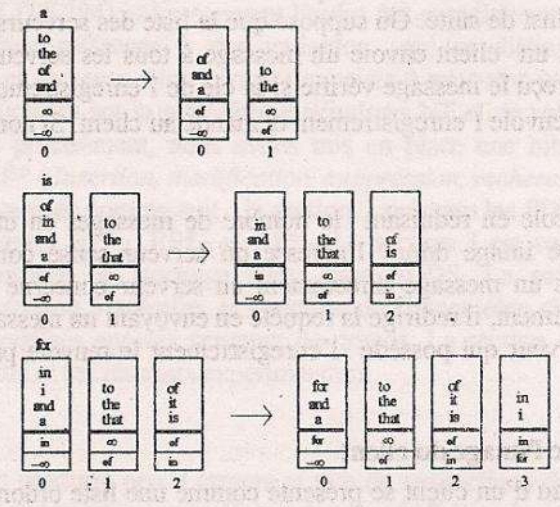


Fig. 2 : Exemple d'évolution d'un fichier SDDS RP*.

En appliquant l'algorithme précédent à l'image d'un client en fonction des IAMs reçus, nous obtenons l'évolution suivante :

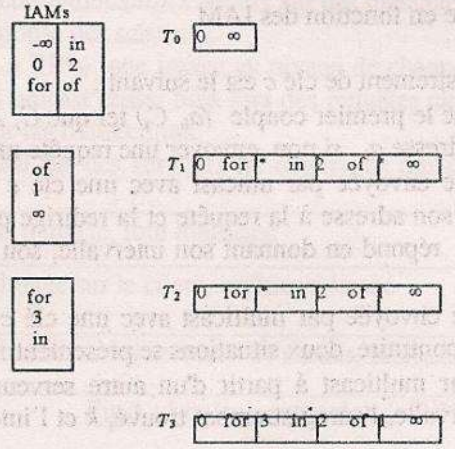


Fig. 3 : Evolution de l'image du client résultant des recherches des clés it, that, in, dans le fichier de la figure précédente.

2.3 SDDS RP*s

Dans RP*$_S$ on améliore à nouveau le protocole par la création d'un index en B-arbre réparti sur des serveurs de nœuds. Cet index représente une image à jour du fichier SDDS. Ainsi les serveurs seront divisés deux catégories: les serveurs de nœud d'index et les serveurs de case. La redirection qui se faisait par multicast dans RP*$_C$ est effectuée ici à l'aide de messages unicast.

2.3.1 Structure de l'index au niveau des serveurs

Chaque nœud peut contenir au plus m $(m >> 1)$ clés, rangées dans l'ordre croissant. Elles sont separées par des pointeurs vers le niveau inférieur. Dans le cas général les pointeurs contenus dans les nœuds non terminaux pointent vers des serveurs de nœud de niveau inférieur; par contre les pointeurs contenus dans les nœuds terminaux font référence aux serveurs de cases. Deux clés successives définissent l'intervalle du nœud ou de la case référencée par le pointeur médian. L'en-tête d'un nœud ou d'une case contient son intervalle et son pointeur père.

2.3.2 Evolution du fichier

L'éclatement d'une case RP^*_S est similaire à celui d'une case RP^*_N sauf ce qui suit :
On désigne par c_m la clé du milieu d'un nœud ou d'une case.

- L'éclatement de la case *0* donne le premier nœud *a* qui représente la racine d'un arbre à deux niveaux avec un intervalle = $(-\infty, +\infty)$, et un pointeur père = *. (voir figure ci-après)
- La case *0* insert le triplet $(0, c_m, 1)$ dans le nœud *a*.
- tout autre éclatement entraîne l'insertion dans *a* de (c_m, r), où *r* est l'adresse de la nouvelle case.
- le pointeur père de la nouvelle case est ensuite relié au nœud *a*.
- si le nœud *a* éclate, alors :
  - * créer un nœud *b* de même père que *a*
  - * créer un nœud *c* avec un intervalle = $(-\infty, +\infty)$ et un pointeur père = *. Le nœud *c* est le père de *a* et *b* et devient la nouvelle racine d'un arbre à trois niveaux.
- le nœud *a* insert dans le nœud *c* le triplet (a, c_m, b).
- tout autre éclatement entraîne l'insertion dans *c* des couples (c_m, r) jusqu'à saturation.

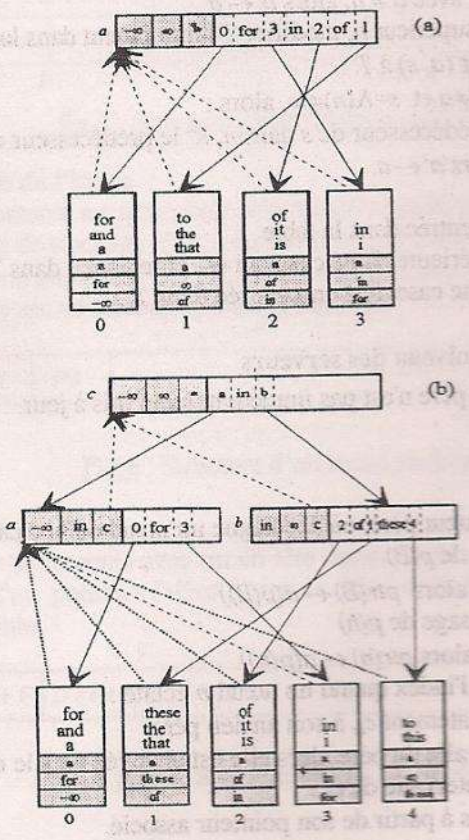


Fig.4 : Evolution d'un fichier RP^*_S [LITWI94].

Après l'éclatement, les pointeurs pères des cases 1,2 et 4 doivent être corrigés pour pointer sur le nœud *b*.

2.3.3 Accès au fichier

Un client qui désire envoyer une requête calcule d'abord l'adresse du serveur de case à partir de son image.

Une case qui reçoit une requête avec une clé c exécute l'algorithme suivant :

- Si c appartient à son intervalle alors exécuter la requête.
- Si non, rediriger la requête vers son nœud père.
- Si c appartient à l'intervalle du père alors envoyer la requête à la case contenant l'enregistrement de clé c.
- Si non, rediriger la requête au niveau supérieur.
- Répéter ce processus jusqu'à ce qu'une feuille soit atteinte.
- La feuille finale renvoie un IAM au client avec la trace du chemin parcouru dans l'arbre par la requête, ce chemin est appelé *IA-Tree*.

2.3.4 Ajustement de l'image du client

Désignons par T la table représentant l'image du client comme dans RP*C. Notons par (a, s) un élément de T où a est un pointeur et s une clé. L'algorithme présenté ci-dessous sera utilisé pour l'ajustement de l'image du client RP*S

On parcourt l'IA-Tree de bas en haut et niveau par niveau :

1. Pour chaque nœud inférieur n et pour chaque couple $(a, s) \in n$, modifier T uniquement dans les cas suivants :
   - (a) Si $s \notin T$, ajouter (a, s) à T en respectant l'ordre de rangement
   - (b) Si non, si $\exists (a, s') \in T$ avec $s' \neq s$, alors $s' \leftarrow s$
   - (c) Si non, si $\exists (a', s) \in T$ avec $a' \neq a$, alors $a' \leftarrow a$
2. Pour chaque (a, s) d'un nœud supérieur n, modifier T uniquement dans les cas suivants :
   - (a) Si $s \notin T$, alors ajouter (a, s) à T
   - (b) Si $\exists (a', s) \in T$ avec $a' \neq a$ et $s = \Lambda(n) \neq \infty$, alors :
     - Soient s' le prédécesseur de s dans n, s'' le prédécesseur de s dans T.
     - Si $s'' \leq s'$, alors $a' \leftarrow a$.

Commentaires:

L'étape 1.(a) : Ajoute une nouvelle entrée dans la table

L'étape 1.(b) : Modifie la borne supérieure d'une case qui est enregistrée dans T

L'étape 1.(c) : Modifie l'adresse d'une case déjà enregistrée dans T

2.3.5 Ajustement de l'index au niveau des serveurs

Quand un nœud éclate, son pointeur père n'est pas immédiatement mis à jour.

Désignons par :

$p(n)$ le nœud père du nœud n

$p(B)$ le nœud père de la case B

$a(X)$ l'adresse de X et $ptr(X)$ son pointeur père, où X désigne un nœud ou une case

- Si une case B reçoit un message de $p(B)$
  alors si $ptr(B) \neq a(p(B))$ alors $ptr(B) \leftarrow a(p(B))$

  Si un nœud père n reçoit un message de $p(n)$
  alors si $ptr(n) \neq a(p(n))$ alors $ptr(n) \leftarrow a(p(n))$

Il y a aussi nécessité d'ajustement de l'index quand un nœud n éclate :

- Il envoie la clé du milieu de l'éclatement c_m à son ancien père.
- Si c_m ne correspond pas à l'intervalle du père alors elle est envoyée vers le niveau supérieur jusqu'à ce que l'on trouve un nœud avec l'intervalle de c_m.
- c_m sera ensuite envoyé vers le bas à partir de son pointeur associé.
- ce nœud envoie finalement un IAM à la case B avec le pointeur père actuel.

3 Organisation interne d'une case SDDS-RP*

Il s'agit dans cette partie d'implémenter les structures de données et les algorithmes décrits précédemment. Dans un premier temps on définit la structure générale d'une case RP*, ensuite la structure des images au niveau des clients et au niveau des serveurs. On insistera particulièrement sur les algorithmes d'éclatement.

3.1 Espace de stockage d'une case SDDS

Physiquement une case SDDS est stockée dans la RAM sous forme de fichier-mémoire ou fichier mappé (memory mapped file) [CUSTE93], [RICHT93]. Un fichier mappé est un mécanisme inter-processus qui est géré comme un fichier classique à l'aide de primitives d'entrée-sorties. Au niveau logique une case est composée d'un en-tête et d'une suite d'enregistrements.

Pour cette étude on considère des clés numériques codées sur *4* octets. Une case SDDS peut contenir au maximum *b* enregistrements *(b >> 1)*, rangés dans l'ordre croissant des valeurs de clés.

3.2 Structure générale d'une case SDDS-RP*

L'espace de stockage d'une case SDDS peut être réparti en trois zones :

- L'en-tête contient l'intervalle de la case, représentée par sa clé maximale Λ et sa clé minimale λ. Pour l'algorithme RP*s, l'en-tête comporte en plus un pointeur vers le nœud père.
- La zone suivante est réservée au stockage de *l'index interne de la case* que nous avons choisi de structurer en B-arbre.
- La dernière zone contient l'ensemble des enregistrements de la case rangés physiquement suivant l'ordre d'arrivée et logiquement chaînés suivant l'ordre croissant des valeurs de clés.

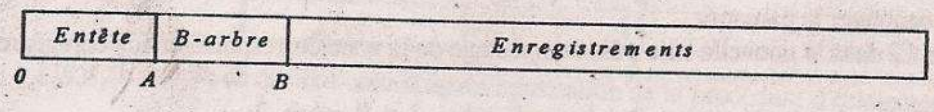


fig.5 : Structure générale d'une case SDDS

3.3 Organisation des nœuds de l'index

Un nœud non terminal peut contenir au maximum *m* enregistrements qui se présentent sous forme de couples *(clé, pointeur)*. L'espace de stockage d'un nœud peut être répartie en deux zones :

- l'en-tête qui est composé d'un pointeur *P* vers le nœud père et du nombre *N* de couples stockés
- la deuxième zone sera réservée au stockage de ses *m* couples.

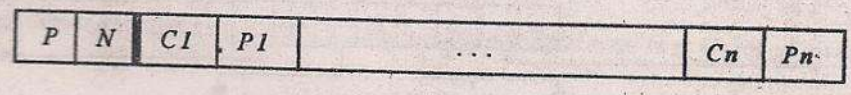


Fig.6 : Structure d'un nœud du B-arbre.

3.4 Organisation des feuilles

Une feuille est une liste d'enregistrements avec un en-tête composé de son pointeur père *P*, du nombre d'enregistrements associés *N*, d'un pointeur *Début F* vers le début de la liste et, d'un pointeur *Début FS* vers le début de la feuille suivante.

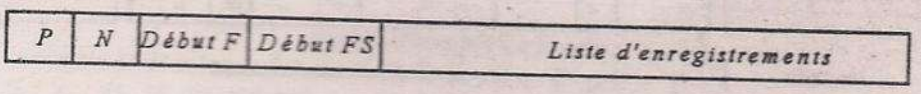


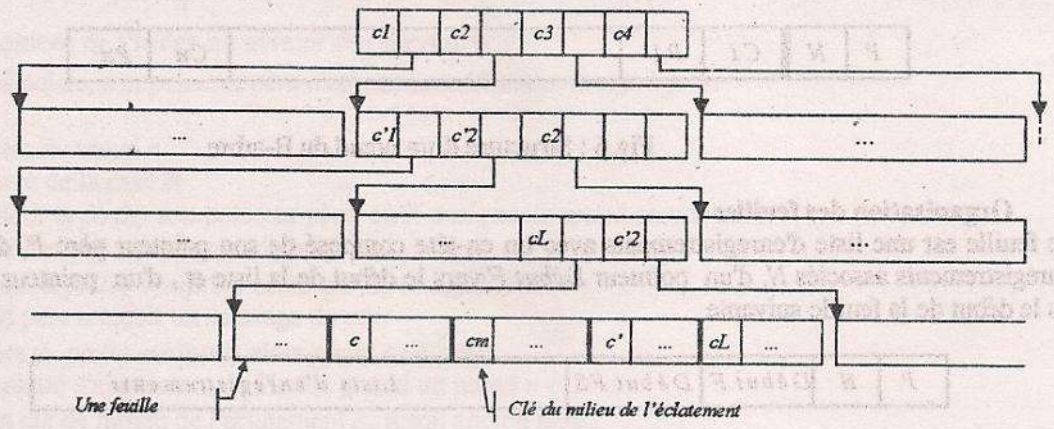
Fig.7 : Structure feuille du B-arbre

4 Eclatement d'une case SDDS

Grâce à l'éclatement des cases, un fichier SDDS peut être étendu, par les insertions, d'un seul site de stockage à n'importe quel nombre de sites. Un éclatement a lieu quand une case déborde. En moyenne *b/2* enregistrements seront transférés vers la nouvelle case. Cette section décrit le processus d'éclatement selon les phases ci-dessous :

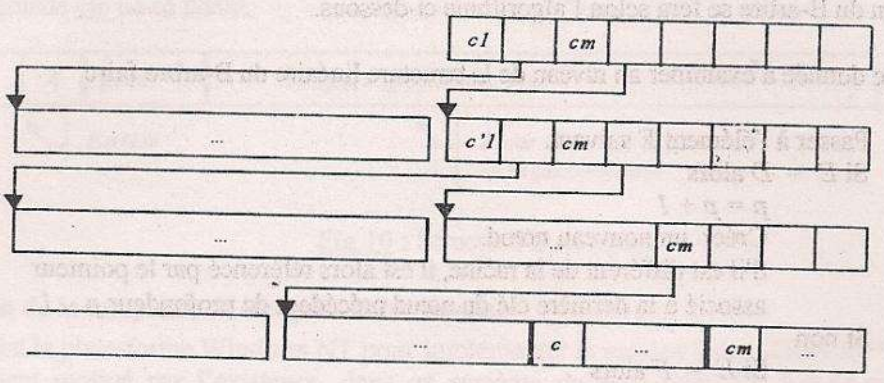
- Recherche de la clé du milieu c_m, et éclatement des enregistrements en deux sous-ensembles *E1 (pour les clés* $\leq c_m$*)* et *E2 (pour les clés* $> c_m$*)*. Soit *ne* le nombre d'enregistrements de la feuille correspondant au couple père *(clé, pointeur)*. Soit *k* un entier positif. On a l'algorithme suivant :

  $k = 0$
  Effectuer un parcours préfixé du B-arbre à partir de la racine.
  Si le couple *(clé, pointeur)* courant correspond au père d'une feuille du B-arbre alors
  $k = k + ne$
  Si $k \geq b/2$ alors
  c_m *appartient à la feuille référencée par le pointeur courantsuivant*
  $k = k - ne$
  Se positionner au début de la liste des enregistrement de la feuille.
  Tant que $k < b/2$ faire
  Début
  $k = k + 1$
  Passer à l'enregistrement suivant
  Fin Tant que
  c_m = *clé* de l'enregistrement courant
  Si non continuer le parcours
  Si non continuer le parcours
- Transfert de *E2* dans la nouvelle case puis compactage de la zone correspondant aux enregistrements déplacés.
- Eclatement du B-arbre en deux *sous-B-arbres* : *B-arbre_1* et *B-arbre_2*.
  Soit *L* la feuille contenant la clé du milieu c_m et p_m son adresse.
  1. Insérer le couple (c_m, p_m) au niveau du nœud père de *L* noté par *Np(L)*
  2. Eclater *Np(L)*. La clé du milieu c_m sera insérée au niveau du nœud père de *Np(L)* qui sera éclaté à son tour à partir de c_m et ainsi de suite.
  3. Poursuivre cette procédure d'éclatement jusqu'à la racine. Après l'éclatement de celle-ci, on obtient deux sous-B-arbres : un *B-arbre_1* et un *B-arbre_2*.



Après éclatement, on obtient :

B-arbre_1



B-arbre_2

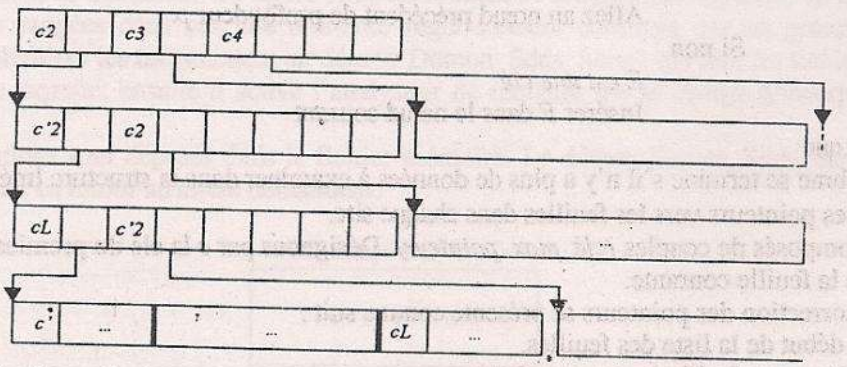


Fig.8 : Exemple de résultat obtenu après application de la procédure d'éclatement.

- Transfert du *B-arbre_2* dans la nouvelle case.
- La méthode choisie consiste à linéariser le *B-arbre* dans un vecteur, avec des délimiteurs de début et fin de nœud. Ce vecteur est facilement transféré sous la forme d'un flot de données. Au niveau de la case réceptrice un module spécifique permet de reconstruire l'arbre initial.

Linéarisation :
Parcourir en préfixe le *B-arbre_2.*
Concaténer les clés rencontrées en les délimitant à laide des symboles *D* et *F* (respectivement de début et de fin de nœud.
Exemple :

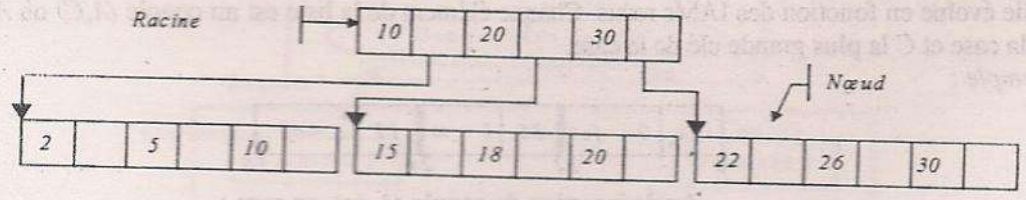


Fig.9 : Structure en B-arbre.

Ce B-arbre sera alors codé selon le format ci-dessous : D = '(' et F = ')'
(10 (2, 5, 10) 20 (15, 18, 20) 30 (22, 26, 30))
Cette structure peut être optimisée en éliminant la deuxième représentation de la *clé maximale.* On obtient alors *(10 (2, 5) 20 (15, 18) 30 (22, 26))*
Reconstruction de l'arbre :
Cette procédure permet de reconstruire le B-arbre au niveau du site récepteur. Considérons la structure linéaire du B-arbre de l'exemple précédent. Désignons par profondeur *p* d'un nœud le nombre de symbole *D* rencontrés avant d'atteindre la première clé de ce nœud à partir du début.

Exemple : Dans l'exemple précédent, la profondeur du nœud (2, 5, 10) est 2.
La reconstruction du B-arbre se fera selon l'algorithme ci-dessous.
$p = 0$
Tant qu'il y a une donnée à examiner au niveau de la structure linéaire du B-arbre faire
Début
Passer à l'élément E suivant.
Si $E = D$ alors
$p = p + 1$
Créer un nouveau nœud.
S'il est différent de la racine, il est alors référencé par le pointeur associé à la dernière clé du nœud précédent de profondeur $p - 1$.
Si non
Si $E = F$ alors
$p = p - 1$
Allez au nœud précédent de profondeur p.
Si non
E est une clé
Insérer E dans le nœud courant
Fin Tant que
L'algorithme se termine s'il n'y a plus de données à examiner dans la structure linéaire.

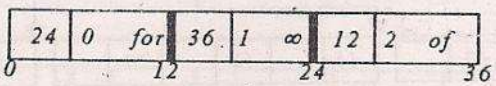
- Mise à jour des pointeurs vers les feuilles dans chaque site.

Les nœuds sont composés de couples *(clé_max, pointeur)*. Désignons par c la clé du premier enregistrement de la feuille courante.
L'algorithme de correction des pointeurs se présente comme suit :
Se positionner au début de la liste des feuilles.
Tant qu'il y a une feuille à examiner faire
Début
Passer à la feuille suivante
Accéder à la clé c de son premier enregistrement
Appliquer la procédure de recherche sur le B-arbre à partir c.
Au niveau du couple *(c, pointeur)* du nœud terminal faire
pointeur = adresse de la feuille courante
Fin Tant que

5 Structure interne de l'image des clients et des serveurs de nœuds

5.1 Structure interne de l'image des clients

Comme nous l'avons déjà précisé, l'image définie au niveau du client se présente comme une liste ordonnée; elle évolue en fonction des IAMs reçus. Chaque élément de la liste est un couple *(A,C)* où A est l'adresse de la case et C la plus grande clé de la case.
Exemple :



Après insertion du couple *(3, in)*, on aura :

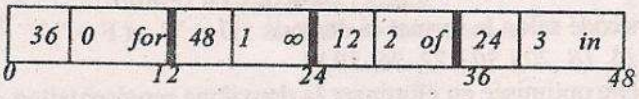


Fig.9 : Exemple d'image client

5.2 Structure interne des serveurs de noeud

Un nœud comporte deux parties :

- l'en-tête qui comprend les clés maximale et minimale du sous arbre et un pointeur vers le nœud père.

- la seconde partie qui contient les couples *(pointeur, clé)*, où *pointeur* fait référence à un nœud fils et *clé* la plus grande clé de ce nœud.

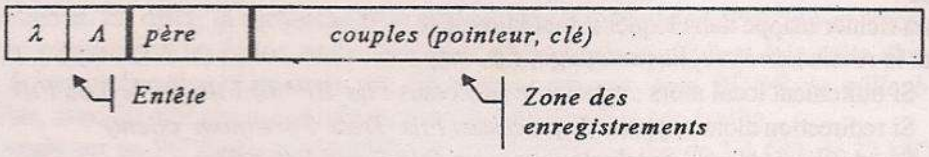


Fig.10 : Structure d'un nœud

6 Réalisation de la maquette de l'organisation interne

Nous avons choisi la plate-forme Windows NT pour implémenter la *version 0* des SDDS-RP*. Ce choix a été essentiellement motivé par l'existence, dans ce système du mécanisme de fichier-mémoire déjà évoqué plus haut dans ce document. La figure ci-après modélise l'architecture logicielle de la maquette.
Les requêtes sont rangées dans une file d'attente régulièrement consultée par un groupe de *threads* [SINHA96]. Ces derniers les transmettent au démon Demon_Sdds_Serv() qui crée un fichier-mémoire et y range la requête extraite; ensuite il active l'analyseur de requêtes qui se charge d'invoquer la routine RP* adéquate.
En retour, les résultats sont déposés dans le fichier-mémoire. Le démon Demon_Sdds_Serv() se charge alors de les expédier en retour au client demandeur.

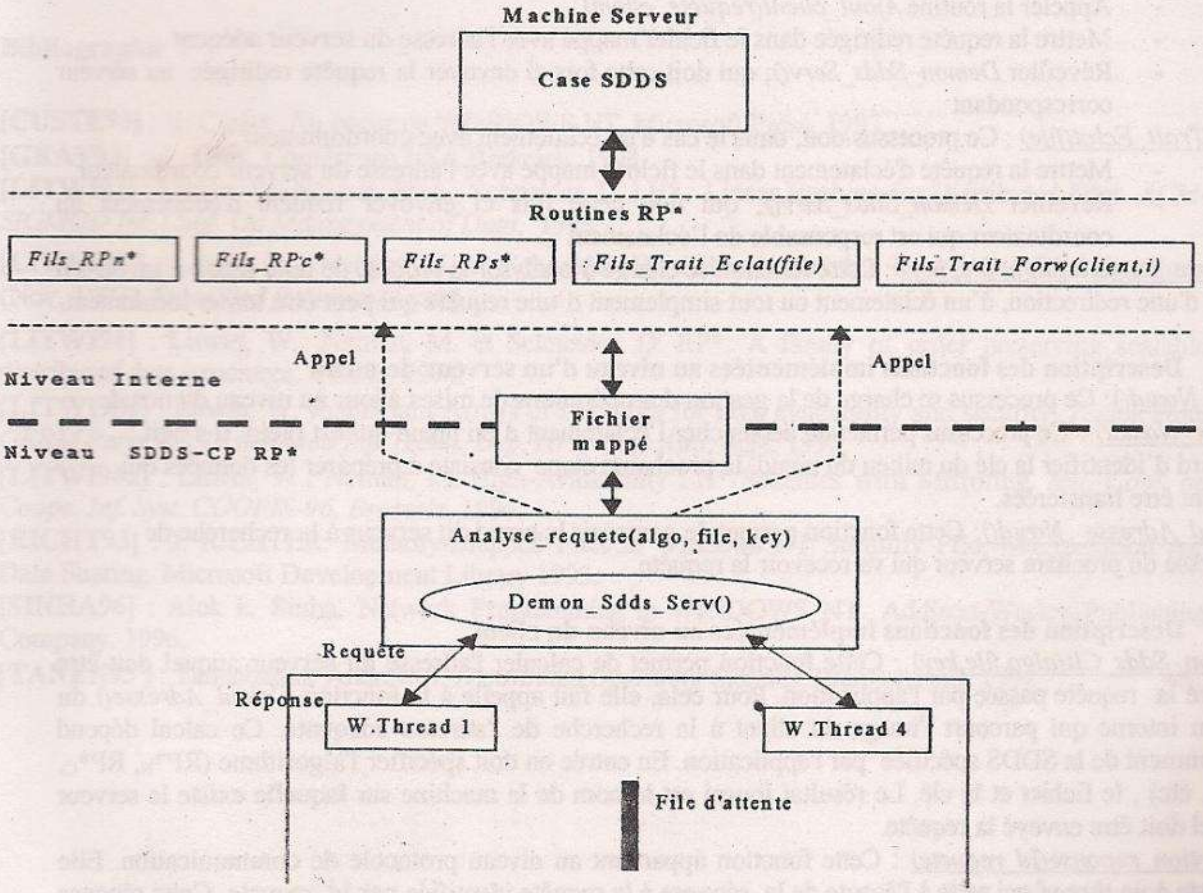


Fig.10 : Description de la maquette de l'organisation interne au niveau du serveur de case

6.1 Description des fonctions implémentée au niveau d'un serveur de case

Demon_Sdds_Serv() : Cette fonction qui sera réalisée au niveau du protocole de communication SDDS-CP RP* permet de :

- Créer un fichier mappé dans lequel il met la requête
- Appeler la routine *Analyse_Requete(algo, file, key)* :
  - Si traitement local alors : appeler le processus *Fils_RP*n(),Fils_RP*c()* ou *Fils_RP*s()*
  - Si redirection alors : appeler le processus *Fils_Trait_Forw(nom_client)*
  - Si éclatement alors : appeler le processus *Fils_Trait_Eclat(file)*

*Fils_RP*n()* : Ce processus doit :

- Faire le traitement adéquat (RP^*_N : *insertion, recherche, suppression*)
- Mettre la réponse dans le fichier mappé
- Réveiller *Demon_Sdds_Serv()*, c'est à dire l'appelant

*Fils_RP*c()* : Ce processus doit :

- Faire le traitement adéquat (RP^*_C: *insertion, recherche, suppression*)
- Mettre la réponse dans le fichier mappé
- Réveiller *Demon_Sdds_Serv()*

*Fils_RP*s()* : Ce processus doit :

- Faire le traitement adéquat (RP^*_S : *insertion, recherche, modification, suppression*)
- Mettre la réponse dans le fichier mappé
- Réveiller *Demon_Sdds_Serv()*

Fils_Trait_Forw(client, i) : Ce processus doit :

- Appeler la routine *Ajout_client(requete, client)*
- Mettre la requête redirigée dans le fichier mappé avec l'adresse du serveur adéquat
- Réveiller *Demon_Sdds_Serv()*; qui doit cette fois ci envoyer la requête redirigée au serveur correspondant

Fils_Trait_Eclat(file) : Ce processus doit, dans le cas d'un éclatement avec coordonnateur :

- Mettre la requête d'éclatement dans le fichier mappé avec l'adresse du serveur coordinateur
- Réveiller *Demon_Sdds_Serv()*; qui doit cette fois ci envoyer requête d'éclatement au coordinateur qui est responsable de l'éclatement

Analyse_Requete(algo, file, key) : Cette fonction se charge d'analyser la requête de telle façon à savoir s'il s'agit d'une redirection, d'un éclatement ou tout simplement d'une requête qui peut être traiter localement.

6.2 Description des fonctions implémentées au niveau d'un serveur de nœud

Maj_Nœud(): Ce processus se charge de la gestion des opérations de mises à jour au niveau du nœud.

Eclate_Nœud() : Ce processus permet de déclencher l'éclatement d'un nœud qui est plein. Il s'agit d'abord d'identifier la clé du milieu du nœud, la prochaine étape consiste à préparer les données qui doivent être transférées.

Calcul_Adresse_Nœud(): Cette fonction permet de parcourir le nœud du serveur à la recherche de l'adresse du prochain serveur qui va recevoir la requête.

6.3 Description des fonctions implémentées au niveau du client

Demon_Sdds_Clt(algo,file,key) : Cette fonction permet de calculer l'adresse du serveur auquel doit être envoyé la requête passée par l'application. Pour cela, elle fait appelle à la fonction *Calcul_Adresse()* du niveau interne qui parcourt l'image du client à la recherche de l'adresse adéquate. Ce calcul dépend évidemment de la SDDS spécifiée par l'application. En entrée on doit spécifier l'algorithme (RP^*_N, RP^*_C, RP^*_S, etc) , le fichier et la clé. Le résultat fourni est le nom de la machine sur laquelle existe le serveur auquel doit être envoyé la requête.

Reception_reponse(Id_requete) : Cette fonction appartient au niveau protocole de communication. Elle consiste à un *thread* qui reste à l'écoute de la réponse à la requête identifiée par Id_requete. Cette réponse peut être reçue d'un serveur différent de celui auquel on a envoyé la requête; c'est à dire dans le cas de redirection. La réponse d'un serveur doit être munie de l'Id_requete. Cette réponse doit être retournée à l'application correspondante. La réponse peut être avec ou sans IAM. Dans le dernier cas, le client doit mettre à jour son image à l'aide de fonction *Trait_IAM()* du niveau interne.

6.4 Tests de validité:

Tous les algorithmes de base *(insertion, suppression, recherche, modification, éclatement)* permettant la manipulation des fichiers RP* ont été implémentés à travers cette maquette. Pour tester leur validité, nous avons mis en place une procédure expérimentale qui s'appuie sur le contrôle de la réversibilité de certaines opérations. En effet, la recherche d'un enregistrement inséré doit aboutir à un succès, celle d'un enregistrement supprimé, à un échec, enfin celle d'un enregistrement déplacé à la suite d'un éclatement, doit aboutir à une redirection. Ces tests ont été réalisés sur une case SDDS de taille $b = 10000$ enregistrements, avec *50* clés par nœud interne et *80* enregistrements par feuille.

Les tests effectués sur un PC Pentium 166 MHz, équipé d'un disque dur de 2,4 GO, de 32 MO de RAM et 512 KO de mémoire cache externe révèlent un rapport entre le temps de recherche sur fichier disque et celui sur fichier-mémoire supérieur à 75. En effet le temps moyen de recherche sur disque local est de 10,6 ms et celui en RAM locale est de 0,14 ms.

7 Conclusion

Nous avons conçu une organisation interne pour les clients et les serveurs de la famille RP*. Nous avons implémenté et testé une bibliothèque de procédures d'accès aux données contenues dans ce type de fichiers. Ce travail s'inscrit dans le cadre de la réalisation de la première implémentation d'un système de fichier distribué et scalable de la famille des SDDS RP*. Il s'agit maintenant de finaliser le développement du protocole de communication permettant aux différents ordinateurs de dialoguer et d'échanger des données en utilisant cette bibliothèque de procédures mise en place .

Bibliographie

[CUSTE93] : H. Custer. Au coeur de WINDOWS NT. Microsoft Press. 1993.

[GRAY93] : J. Gray. Conférence USA Berkeley, 1993.

[LITWI93] : Litwin, W. Neimat, M-A., Schneider, D. LH* : Linear Hashing for Distributed Files. *ACM-SIGMOD Int. Conf. On Management of Data*, 1993.

[LITWI93'] : Litwin, W., Neimat, M-A., Schneider, D. LH*: A Scalable Distributed Data Structure (Nov. 1993). *Submitted for journal publ.*

[LITWI94] : Litwin, W., Neimat, M. et Schneider, D. RP*: A family of order preserving scalable distributed data structures. VLDB, 1994.

[LITWI95a] : Litwin, W. Redundant Arrays of LH* files for high availability and security. *Techn. Note GERM Paris 9 & Distributed Inf. Techn. Dep. HPL Palo Alto*, Sept. 1995.

[LITWI96a] : Litwin, W., Neimat, M. High-Availability LH* Schemes with Mirroring. *Intl. Conf. on Coope. Inf. Syst. COOPIS-96, Brussels*, 1996.

[RICHT93] : J. RICHTER. Memory-Mapped Files in Windows NT Simplify File, Manipulation and Data Sharing. Microsoft Development Library 1993.

[SINHA96] : Alok k. Sinha. Network Programming in WINDOWS NT. Addison-Wesley Publishing Company. 1996.

[TANEN95] : Tanenbaum, Andrew S. *Distributed Operating Systems*. Prentice Hall, 1994.