REPUBLIQUE DU SENEGAL

UNIVERSITE CHEIKH ANTA DIOP DE DAKAR

FACULTE DES SCIENCES ET TECHNIQUES

Département Informatique

Mémoire de DEA

Option : Bases De Données

Sujet

Mise en œuvre de l'architecture de communication des Structures de Données Distribuées et Scalables RP*N et RP*C

Présenté et soutenu par

Yakham Ben Abdel Kader Ndiaye

le 27 juin 1998

Sous la direction de :

Witold Litwin	Professeur Université Paris 9 Dauphine

Devant le jury composé de :

M.M. Chérif Badji	Professeur
Gérard Lévy	Professeur
Richard Emilion	Chargé d'enseignement
Gérard Lambert	Maître de conférence
Samba Ndiaye	Maître assistant
Tidiane Seck	Maître assistant
Jules-Raymond Tapamo	Maître assistant
Boubacar Barry	Assistant

Mise en œuvre de l'architecture de communication des SDDS (Structures de Données Distribuées et Scalables) RP^*_N et RP^*_C

Mémoire de DEA

Option : Bases De Données

Présenté et soutenu par

Yakham Ben Abdel Kader Ndiaye

le 27 Juin 1998 à 10H

Sous la direction de :

Witold Litwin	Professeur, Université Paris 9 Dauphine

Devant le jury composé de :

M.M. Chérif Badji	Professeur
Gérard Lévy	Professeur
Richard Emilion	Chargé d'enseignement
Gérard Lambert	Maître de conférence
Samba Ndiaye	Maître assistant
Tidiane Seck	Maître assistant
Jules-Raymond Tapamo	Maître assistant
Boubacar Barry	Assistant

Résumé

L'informatique d'aujourd'hui est centré sur le réseau et non plus sur l'ordinateur isolé. C'est là une évolution majeure. La généralisation des réseaux et l'irruption d'Internet en sont des preuves éclatantes. Des avancées majeures ont été accomplies dans les performances des réseaux (standardisation de TCP/IP, Fast Ethernet, ATM, commutateurs larges bandes etc...). Un nouveau concept est apparu: celui de multi-ordinateur, offrant des capacités quasi illimités de calcul, de mémoire vive et de stockage. Les applications actuelles n'arrivent pas à tirer profit des capacités supplémentaires en mémoire vive. Une nouvelle famille de structures de données appelées SDDS a vu le jour pour palier cette insuffisance dans le domaine de la gestion des fichiers. En effet, dans un multi-ordinateur, il est plus rapide d'accéder à une mémoire vive distante qu'au disque local (100 µs contre 10 ms pour les réseaux Ethernet). Plusieurs de ces SDDS ont été proposées par Litwin, Neimat et Schneider. L'équipe de recherche sur les SDDS de Dakar dirigée par Litwin a en charge l'implémentation de la famille des SDDS RP*.

Le travail fait dans ce mémoire a permis l'implantation d'un prototype de communication pour les fichiers distribués RP*. Il s'agit de construire une couche supplémentaire en se basant sur les protocoles TCP et UDP pour permettre l'échange rapide et fiable de données sur un multi-ordinateur en utilisant les technologies récentes comme le multicasting.

Ce travail, à cheval entre les bases de données et les réseaux, a permis de nous familiariser avec la programmation réseau avec les sockets Windows, la programmation parallèle, les problèmes de synchronisation avec les Threads et la manipulation de données distribuées.

IN MEMORIAM

A mon premier professeur d'Informatique à l'Université de Saint-Louis, feu Ousmane SECK

A mon ami feu Emile Victor Bock.

Mes remerciements sincères vont :

Au professeur Gérad LEVY, responsable du DEA d'Informatique de l'Université Cheikh Anta Diop de Dakar.

Au professeur Witold LITWIN, de l'Université Paris 9 Dauphine, pour avoir accepté d'encadrer ce mémoire, pour ses encouragements et son soutien.

A Tidiane SECK, maître-assistant à l'Ecole Supérieur Polytechnique de Dakar, et à Samba NDIAYE, maître-assistant à l'Université Cheikh Anta Diop de Dakar
pour l'encadrement local de ce travail, leurs conseils judicieux, leur disponibilité et leurs encouragements.

A tous les enseignants du Département de Mathématiques et d'Informatique de l'Université Cheikh Anta Diop de Dakar

A tous les enseignants de l'UFR de Mathématiques Appliquées et d'Informatique de l'Université Gaston Berger de Saint-Louis

A l'ensemble du personnel de la Société INTERACTIVE pour leur compréhension et leur encouragement sans faille.

A ma famille et à tous mes amis.

SOMMAIRE

Introduction ____ 1

Chapitre 1
Les Structures de Données Distribuées SDDS

I. Les multiordinateurs ____ 3

II. Les Structures de Données Distribuées SDDS ____ 4

III. Principes des SDDS RP* ____ 6

- III. 1 Structure et évolution du fichier ____ 6
- III. 2 Algorithme d'éclatement d'une case ____ 7
- III. 3 Opérations sur le fichier ____ 8

IV. Les Algorithmes $RP*_N$ ____ 8

- IV. 1 Envoi d'une requête RP*N par un Client ____ 8
- IV. 2 Traitement d'une requête $RP*_N$ par un serveur ____ 9

V. Les Algorithmes $RP*_C$ ____ 9

- V. 1 Structure de l'image du client ____ 10
- V. 2 Envoi d'une requête $RP*_C$ par un Client ____ 10
- V. 3 Traitement d'une requête $RP*_C$ par un serveur ____ 10
- V. 4 Ajustement de l'image du client ____ 11

Chapitre 2
Protocole de Communication : SDDS RP_*N et RP_*C

I. Traitement des messages SDDS-RP* ____ 12

- I. 1 Messages de données ____ 13
- I. 2 Messages de service ____ 13
  - I. 2-1 Messages de redirection (les forwards) ____ 13
  - I. 2-2 Messages d'accusé de réception ____ 14

II. Protocole de communication RP^*_N 15

II. 1 Protocole d'insertion 15

II. 2 Protocole de Suppression 16

II. 3 Recherche d'un enregistrement 17

II. 4 Recherche par intervalle 18

II. 5 Protocole d'éclatement 19

III. Protocole de communication RP^*_C 22

III. 1 Protocole d'insertion 22

III. 2 Protocole de Suppression 25

III. 3 Recherche d'un enregistrement 27

IV. Description des messages 31

IV. 1 Insertion 31

IV. 2 Suppression 32

IV. 3 Recherche d'un enregistrement 32

IV. 4 Recherche par intervalle 32

IV. 5 Eclatement 33

Chapitre 3
Outils techniques pour la mise en oeuvre du prototype

I. Interface de communication entre processus distants 34

I. 1 Caractéristiques d'une socket 35

I. 2 Communication par datagrammes 38

I. 3 Communication en mode connecté 39

II. La transmission multipoint (Le multicast) 42

II. 1 Implémentation du multipoint IP 42

II. 2 Le routage multipoint 43

II. 3 Protocoles multicast 44
II. 4 Supports requis 45
II. 5 Multipoint sur réseau Ethernet 45
II. 6 Emission de datagrammes multicast 45
II. 7 Réception d'un datagramme multicast 46
II. 8 Exemple de réseau multicast : Le Mbone 46
III. Programmation multitâche 47
III. 1 Processus et Threads 47
III. 2 Synchronisation de l'Exécution des Thread 48
III. 3 Mémoire Partagée 49

Chapitre 4
Description du Prototype de communication SDDS RP*

I. L'Architecture Client/Serveur 50
II. L'application SDDS-RP* / Client 51
III. L'application SDDS-RP* / Serveur 54
IV. Mesure de performances 58

Conclusion 59

Références 60

INTRODUCTION

Au cours des dernières années, la vitesse des CPU a constamment augmenté. Parallèlement, la performance des réseaux a connu des améliorations importantes avec des débits de plus en plus élevés. Mais, les disques durs, limités par des contraintes mécaniques, n'ont pas atteint le même succès du point de vue temps d'accès. L'accès aux données sur une mémoire distante est maintenant plus rapide que la lecture de données stockées sur un disque local.

Des recherches avancées sont menées pour mieux exploiter la puissance de calcul d'un ensemble d'ordinateurs interconnectés à travers des réseaux à haut débit. De telles configurations existent déjà dans plusieurs organisations. D'habitude, cela consiste en quelques ordinateurs interconnectés, voire des milliers dans les grandes organisations [NOW96].

Des termes sont apparus pour désigner les machines organisées de la sorte : mémoires distribuées, multiordinateur. La capacité de traitement parallèle et les capacités de stockage RAM et disque cumulées des machines formant un multiordinateur sont impressionnantes et même supérieures à la performance des ordinateurs gros systèmes.

Les structures de données utilisées par les fichiers sont un composant important d'un multiordinateur. En effet, les multiordinateurs permettent de lancer des traitements distribués et parallèles qui ne peuvent pas être supportés par les structures de données traditionnelles. Les fichiers distribués peuvent être très volumineux et une de leurs qualités essentielles doit être la scalabilité qui consiste à pouvoir obtenir le même temps de réponse quand le nombre d'enregistrements augmente.

Une classe de structures de données est apparue pour ouvrir de nouvelles perspectives sur la gestion des fichiers sur les multiordinateurs : les Structures de Données Distribuées et Scalables (SDDS). Elles doivent permettre de construire des fichiers dynamiques qui seront stockés au niveau de la mémoire distribuée d'un multiordinateur.

Dans un premier temps, des SDDS basées sur le hachage ont été proposées : LH* et ses variantes, DDH [LITWI80] [LITWI93] [LITWI93a] [LITWI80] [LITWI96a]. Elles constituent une généralisation de l'algorithme de hachage. Comme le hachage ne préserve pas l'ordre des enregistrements, alors les structures de données ordonnées telles que les arbres-B sont plus rapides si le fichier doit supporter des requêtes par intervalle ou un parcours transversal des enregistrements. Une famille de SDDS : RP* (Range Partitioning), a été proposée pour construire des fichiers distribués qui préservent l'ordre des enregistrements. Elle est composée de trois variantes : RP*N, RP*C et RP*S [LITWI94] [LITWI95b].

L'objet de ce travail est l'étude détaillée des algorithmes RP*N, RP*C, la conception d'un protocole de communication et la réalisation d'un prototype de système de communication pour les SDDS RP*N et RP*C.

Les principes de base des SDDS et les algorithmes RP*N et RP*C feront l'objet du premier chapitre. Le deuxième chapitre sera consacré à une description détaillée des messages échangés dans le protocole de communication. Les choix techniques pour la mise en œuvre du prototype SDDS RP* et la description de son architecture Client/Serveur seront présentés en dernier lieu.

Chapitre 1
Les Structures de Données Distribuées SDDS

I. Les multiordinateurs

Un multiordinateur est une collection d'ordinateurs sans mémoire partagée reliés par un bus de type LAN ou WAN. C'est un concept qui a l'avantage de s'appuyer sur l'existant. La puissance théorique cumulée des ressources en calcul et en mémoire des multiordinateurs est impossible à atteindre pour un ordinateur traditionnel. Ces performances ont favorisé l'apparition de plusieurs projets de recherche en Informatique dont le but est de produire des logiciels permettant d'exploiter toutes leurs potentialités. L'un des axes de recherche est la construction de système de gestion de fichiers dynamiques résidant entièrement sur la RAM distribuée du multiordinateur.

Les tendances de l'évolution du hardware et des réseaux, notamment la performance d'accès à la mémoire vive distante par rapport au disque local (cf. Figure I.1), encouragent de plus en plus la conception d'applications pour les multiordinateurs. Les applications potentielles sont les SGBD, les calculs hautes performances, les programmes à haute disponibilité ou le temps réel.

A part les réseaux, tous les logiciels systèmes sont à (re)faire, notamment les SGF pour mieux utiliser la RAM distribuée et offrir des fichiers extensibles en mémoire.

Temps d'accès aux données				
Ressource	RAM local	RAM distant (réseau gigabit)	RAM distant (Ethernet)	Disque local
Temps d'accès	100 nsecs	1 µsecs	100 µsecs	10 msec
En proportion	1 mn	10 mn	2 heures	8 jours

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

Caractéristiques réseaux des multiordinateurs

On considère un réseau local typique. Des machines sont reliées à un câble appelé segment par leurs contrôleurs. Les segments sont reliés entre eux par des routeurs qui transmettent les messages externes. Les postes envoient des messages qui sont souvent découpés en paquets. Chaque message est écouté par tous les contrôleurs du segment. Il existe trois types de messages :

- **Point-à-point** : ce message est retenu par un contrôleur unique qui le délivre à son site.
- **Broadcast** : ce type de message porte une adresse reconnue par tous les contrôleurs du réseau. Ainsi, tous les sites du réseau reçoivent ce message.
- **Multicast** : ce type de message porte une adresse reconnue seulement par une partie des contrôleurs du segment. Seuls les contrôleurs concernés délivrent alors ce message à leur site respectif. Les autres sites ne sont pas interrompus par leurs contrôleurs.

Un message multicast a le même temps de parcours du réseau qu'un message point-à-point. Il peut délivrer une opération à plusieurs sites en un temps inférieur à celui consécutif à l'utilisation de plusieurs messages point-à-point. Le multicast est particulièrement adapté au cas où une opération devrait être exécutée en parallèle sur un groupe de sites. Aussi ce procédé est-il préférable quand une opération est destinée à un groupe de sites sans interférence aucune sur les autres. La transmission multicast sera traitée plus en détail au chapitre 3.

II. Les Structures de Données Distribuées SDDS

Une structure de données distribuées classique a un point d'accès unique. Un point unique de calcul d'adresse traite les requêtes des différents clients répartis sur le réseau.

La transposition de ces structures de données sur les multiordinateurs met en exergue le problème du point d'accumulation inhérent au calcul d'adresse centralisé. En plus, ceci limite les performances d'accès et rend le système vulnérable aux pannes. La duplication du calcul d'adresse sur les machines qui accèdent aux données distribuées exige la mise à jour

synchrone des tables d'adressage en cas d'éclatement d'une case. Cette action est impossible du fait de l'aspect aléatoire de la connexion des postes clients aux serveurs.

Les Structures de Données Distribuées et Scalables (SDDS) constituent une famille de structures de données définies spécifiquement pour les multiordinateurs (cf. figure II.1). Elles stockent les données sur des sites désignés Serveurs et d'autres sites appelés 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.

Un fichier SDDS débute sur un seul site de stockage et peut être étendu par des insertions à un nombre quelconque de sites. Ceci rend sa capacité de stockage potentiellement illimitée.

Les données d'une SDDS résident en mémoire vive distribuée du multiordinateur. En effet, le temps d'accès des données en mémoire vive distribuée est plus court que celui aux données stockées sur disque. 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. Ces caractéristiques doivent assurer aux SDDS des performances de traitement inconnues des structures de données classiques.

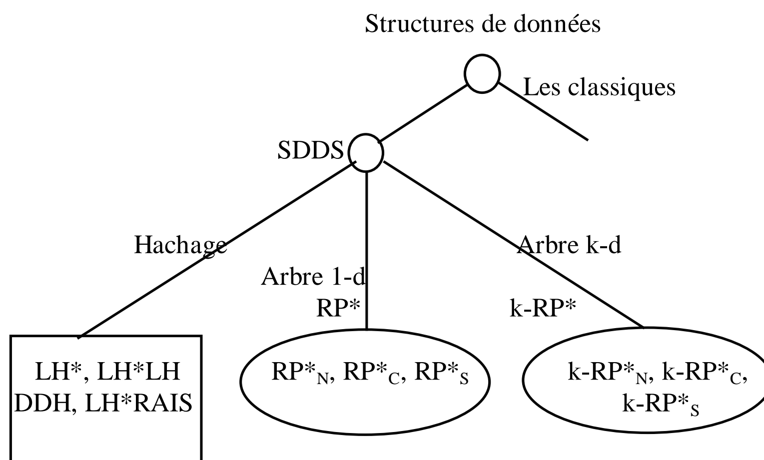


Figure II.1 : Les structures de données distribuées et scalables (SDDS)

Les Règles de base des SDDS

Les données sont sur des sites serveurs.

Il n'y a pas de répertoire central d'accès.

Les mises à jour de la structure d'une SDDS ne sont pas envoyées aux clients d'une manière synchrone.

Un client peut faire des erreurs d'adressage par suite d'une image inadéquate de la structure de SDDS.

Chaque serveur vérifie l'adresse de la requête et l'achemine vers un autre serveur si une erreur est détectée. L'ensemble des renvois à la suite d'une erreur d'adressage ne se fait qu'en quelques messages.

Le serveur adéquat envoie alors un message correctif (IAM - Image Adjustment Message) au client ayant commis l'erreur d'adressage.

Le client ajuste son image pour ne plus faire la même erreur. Les IAM font converger toute image d'un client vers celle actuelle.

III. Principes des SDDS RP*

Les SDDS RP* constituent une généralisation des algorithmes de hachage. Leur but est d'offrir un fichier distribué qui préserve l'ordre des enregistrements. Un fichier peut débuter sur une machine et s'étendre par des insertions sur un nombre quelconque de machines. Le fichier peut être manipulé par plusieurs clients répartis sur le multiordinateur et supporte les requêtes parallèles.

III. 1 Structure et évolution du fichier

Un enregistrement comporte un champ-clé et des champs non-clé. Le champ clé identifie l'enregistrement. L'espace des champs-clé est totalement ordonné. Les enregistrements sont groupés en paquets appelés buckets. Un bucket ou case SDDS a une capacité de b enregistrements. Les enregistrements sont répartis dans les différentes cases suivant la valeur de leur champ-clé. Logiquement, les enregistrements sont rangés suivant l'ordre ascendant des clés dans une case.

La définition d'une structure de données pour l'organisation interne d'une case a fait l'objet d'un mémoire de DEA réalisé par un chercheur de l'équipe SDDS de Dakar [DIENE98].

Chaque case est munie d'un en-tête contenant deux valeurs λ et Λ qui sont respectivement la borne inférieure et la borne supérieure des enregistrements pouvant être contenus dans la case. [λ, Λ] est appelé intervalle ou portée de la case. Une case de portée [λ,Λ] contiendra les enregistrements de clé c dont $\lambda < c \leq \Lambda$. A la création, un fichier RP* comporte une case unique notée case 0 avec $\lambda = -\infty$ et $\Lambda = +\infty$.

Une adresse multicast peut être assignée à cette case aussi bien qu'à toutes les autres cases potentielles du fichier. Toutes les insertions d'enregistrements vont à la case 0. Cette case éclatera en deux dès que son contenu aura atteint la capacité permise pour une case. Il en résulte la création d'une nouvelle case numérotée case 1 qui recevra la moitié des enregistrements de la case 0. La figure suivante illustre l'évolution d'un fichier SDDS RP* avec une capacité des cases fixée à 4. Les enregistrements sont symbolisés par leur champ-clé.

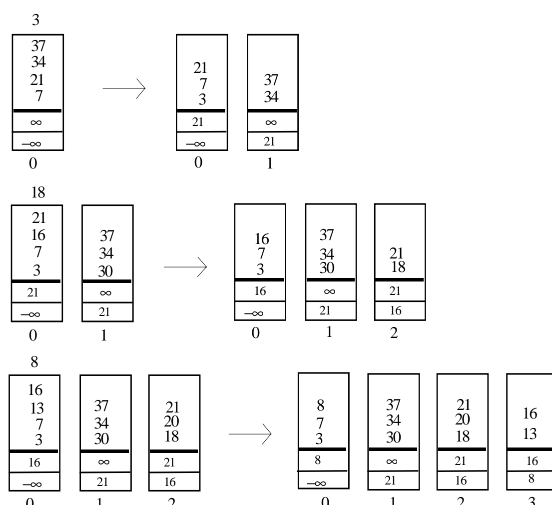


Figure 2 : Evolution du fichier à la suite d'insertions d'enregistrements

III. 2 Algorithme d'éclatement d'une case

Si le nombre d'enregistrements d'une case i atteint la capacité b, alors toute tentative d'insertion dans cette case entraîne son éclatement. L'éclatement consiste en trois opérations : création d'une nouvelle case j, migration de la moitié des enregistrements de la case i vers la case j, modification de la portée de la case i et détermination de celle de la case j. Si cm est la clé de l'enregistrement du milieu de la case en débordement, alors on obtient :

Case i[λ, Λ] => Case i[λ, *cm*] et Case j[*cm*, Λ].

Ainsi, chaque éclatement d'une case affine le partitionnement de l'espace des clés.

Les étapes de l’algorithme

1) Déterminer Cm la clé de l’enregistrement du milieu de la case en débordement.
2) Créer une nouvelle case j.
3) Déterminer l’en-tête de la case J :
   $\lambda j \ <= Cm$
   $\Lambda j \ <= \Lambda$
   Copier dans la case j les enregistrements de la case i dont la clé c > Cm.
4) Modifier l’entête de la case i
   $\lambda i \ <= \lambda$
   $\Lambda i \ <= Cm$
   Effacer les enregistrements de la case i dont la clé c > Cm.

III. 3 Opérations sur le fichier

Le fichier est manipulé par des requêtes émises à partir des sites clients. Une requête peut porter sur un enregistrement, un intervalle d’enregistrements ou un ensemble d’enregistrements qui vérifient une condition sur un champ non-clé. Une requête du type recherche, modification ou suppression d’un enregistrement est dite simple. Une requête par intervalle correspond à la recherche des enregistrements dont les clés sont comprises dans l’intervalle de recherche. Une recherche est dite générale quand la condition porte sur un champ non-clé. Un client peut aussi effectuer un parcours transversal du fichier qui consiste à examiner séquentiellement tous les enregistrements suivant l’ordre croissant ou décroissant des clés.

IV. Les Algorithmes RP*N

IV. 1 Envoi d’une requête RP*N par un Client

Une requête est envoyée par un message multicast avec le nom du fichier. Ce message est reçu par toutes les cases et chacune vérifie si la clé appartient à sa portée. Seul le serveur concerné répond.

Si le client reçoit une réponse avec uniquement les bornes inférieure et supérieure d'une case, cela équivaut à une requête infructueuse. Les réponses des serveurs aux clients s'effectuent par des messages point-à-point.

Dans le cas d'une requête par intervalle, la requête se termine si l'union des portées des cases qui ont répondu recouvre l'intervalle de recherche, ou bien le client se contente des réponses obtenues à l'expiration d'un time-out.

IV. 2 Traitement d'une requête RP*N par un serveur

L'algorithme d'éclatement partitionne l'espace des clés de telle sorte que chaque clé ne peut appartenir qu'a une seule portée. Seule donc la case concernée traite la requête et renvoie le résultat au client avec sa portée.

Une case qui reçoit une requête générale effectue l'opération sur tous les enregistrements concernés par la condition. Dans le cas d'une requête par intervalle, chaque case vérifie l'intersection de sa portée avec l'intervalle de recherche et traite la requête si l'intersection n'est pas vide.

V. Les Algorithmes RP*C

L'algorithme RP*C est une reprise de RP*N auquel on ajoute une image du fichier au niveau des sites clients pour réduire l'usage du multicast. L'image du fichier est une collection d'intervalles et d'adresses de sites Serveurs qui traduit la répartition des enregistrements sur les cases et les serveurs qui les hébergent. Un client peut faire une erreur d'adressage par suite d'une image inadéquate de la structure du fichier SDDS. Dans ce cas, le serveur qui a reçu la requête l'achemine par multicast vers les autres cases. Le serveur adéquat envoie alors un message correctif (IAM) au client. Le client ajuste son image pour ne plus faire la même erreur. Les IAM qui sont encapsulés dans les messages de réponse font converger toute image d'un client vers celle actuelle.

V. 1 Structure de l'image du client

L'image du client est une table dynamique *T[0 , 1, …]*. Chaque élément *T[i]* de cette table définit l'adresse d'une case et son intervalle. Logiquement, la table T est une liste ordonnée de couples *T[i] = (A, C)* avec :

- *A* : Adresse d'une case du fichier SDDS

  On pose *A* = * (une adresse multicast) si elle correspond à une adresse inconnue.

- *C :* Clé maximale Λ(A) que peut contenir la case A.

Initialement $T = [(0, \infty)]$, elle évolue en fonction des messages correctifs (IAM) reçus qui entraînent l'insertion ou la mise à jour de couples.

V. 2 Envoi d'une requête RP*C par un Client

Une requête sur un enregistrement de clé *c* est exécutée de la manière suivante :

Le client parcourt les couples *t = (A, C)* de son image et recherche le premier dont la clé est immédiatement supérieure à la clé c.

- Si $A(t) \neq *$ alors envoyer une requête unicast au *bucket* d'adresse *A*.
- Sinon, envoyer une requête en multicast.

V. 3 Traitement d'une requête RP*C par un serveur

Chaque serveur qui reçoit une requête vérifie si la clé qui y est contenue appartient à sa portée :

1) Si la clé n'appartient pas à sa portée :

   Si le message est multicast, alors sans suite ;

   Si le message est unicast, alors renvoyer la requête plus l'adresse et la portée de la case en multicast aux autres serveurs : il s'agit de la redirection d'une requête.

2) Si la clé appartient à sa portée :

   Si c'est un message redirigé, alors envoyer la réponse, l'adresse et la portée de la case intermédiaire, l'adresse et la portée de la case au client.

   Sinon envoyer la réponse plus l'adresse et la portée de la case au client.

V. 4 Ajustement de l'image du client

La réponse à une requête contient un champ IAM qui permet au client de corriger son image du fichier (cf. figure V. 4-1). L'IAM se présente sous forme d'un ou deux triplets (λ, a, Λ) : [λ,Λ], la portée de la case serveur qui a traité la requête et « a » son adresse. L'ajustement de l'image du client se fait donc de manière asynchrone suivant l'algorithme ci-dessous :

(a) S'il n'existe pas un élément t appartenant à T avec $C(t) = \lambda$ et $\lambda \neq -\infty$

alors insérer $(*, \lambda)$ dans T.

(b) S'il existe un élément t appartenant à T avec $C(t) > \Lambda$

alors : si $C(t) = +\infty$ alors $t = (a, \Lambda)$ et ajouter $(*, +\infty)$ dans T.

si $C(t) < +\infty$ alors $t = (a, \Lambda)$.

(c) S'il existe un élément t appartenant à T avec $t = (*, \Lambda)$

alors $t = (a, \Lambda)$.

(d) S'il n'existe pas d'élément $t = (a, \Lambda)$ appartenant à T

alors insérer (a, Λ) dans T.

C0	C1	C2	C3
8 7 3	37 34 30	21 20 18	16 13
8	∞	21	16
-∞	21	16	8

Etat actuel du fichier

Image initiale : T0 [C_0, + ∞]

Clé recherchée		IAM		évolution image du client
7	⟶	$C_0(-\infty,8)$	⟶	T1 [C_0, 8] [* , + ∞]
19	⟶	$C_2(16,21)$	⟶	T2 [C_0, 8] [*, 16] [C_2, 21] [* , + ∞]
34	⟶	$C_1(21,+\infty)$	⟶	T3 [C_0, 8] [*, 16] [C_2, 21] [C_1, + ∞]
11	⟶	$C_3(8,16)$	⟶	T4 [C_0, 8] [C_3, 16] [C_2, 21] [C_1, + ∞]

Figure V. 4-1 Evolution du fichier à la suite de la recherche des enregistrements(7,19,34,11)

Chapitre 2
Protocole de Communication : SDDS RP*N et RP*C

Le but de ce chapitre est de proposer un protocole de communication pour les algorithmes RP*N et RP*C. Il s'agit de construire une bibliothèque de messages pour toute la famille des SDDS. Pour pouvoir utiliser les mêmes messages quel que soit l'algorithme ou le protocole, chaque message aura en paramètre toutes les informations qui sont susceptibles d'être utilisées.

I. Traitement des messages SDDS-RP*

Les différents types de messages utilisés dans le protocole de communication SDDS-RP* sont organisés suivant une hiérarchie (cf. FigureI.1). Chaque "fils" hérite des caractéristiques de ses "ancêtres". Il s'agit pour nous de développer l'arbre à partir de RP*C. La branche LH* est développée à Paris Dauphine par le GERM [SOULE95] [SAHLI96].

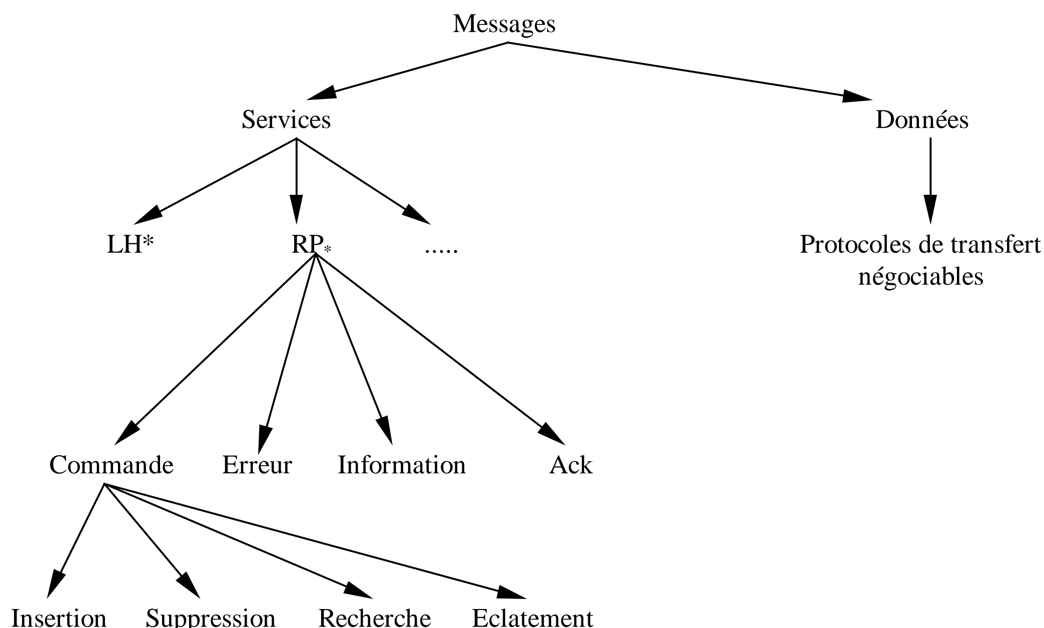


Figure I.1 : Hiérarchie des messages.

Les deux grandes classes de messages sont les **messages de données** utilisés pour le transfert des données et les **messages de service** utilisés pour des préavis de certaines transactions et pour les commandes d'administration du système. Les messages de commande peuvent différer selon la SDDS employée pour générer le fichier.

I. 1 Messages de données

Pour pouvoir transférer des données de la manière la plus efficace, il semble important de pouvoir choisir le protocole de transport de données le plus adapté. Une négociation préalable de ce protocole est faite, avant toute transaction, de la manière suivante : le demandeur envoie la liste des protocoles dont il dispose et l'émetteur choisit le protocole correspondant le mieux au type de données à transférer en fonction de ceux dont il dispose. Cette présente version de l'architecture de communication prend en charge les protocoles TCP et UDP : UDP pour les datagrammes, TCP pour les flux de données.

Le transfert de données se fait dans le sens client vers serveur pour les opérations d'insertion et de modification d'enregistrement. Il se fait dans le sens serveur vers client en réponse à une requête de recherche. Les serveurs s'échangent des données lors de l'éclatement d'une case.

I. 2 Messages de service

Les messages de service utilisent le protocole UDP. Ces messages contiennent les différentes commandes fonctionnelles (les informations sur les opérations à exécuter). Ces messages peuvent être unicast ou multicast en fonction du type d'opération souhaitée. Un message unicast est envoyé à un destinataire unique et le multicast est destiné à un groupe particulier de machines.

I. 2-1 Messages de redirection (les forwards)

RP*N : Il n'existe pas de messages de redirection de serveur à serveur, les différentes requêtes d'un client sont envoyées simultanément à tous les serveurs et seul le concerné répond.

RP*C : Grâce à l'utilisation de messages courts contenant des informations sur l'opération et l'émetteur, en préavis des différentes requêtes, les redirections de serveur en serveur n'ont pas à véhiculer les données (pour une requête en insertion, par exemple). C'est le serveur pouvant accepter la requête qui envoie au demandeur un préavis de transaction. La redirection arrive si un serveur reçoit un message unicast avec une clé n'appartenant pas à sa portée. Il insère alors sa portée et l'adresse de l'expéditeur dans le message reçu pour former le foward à

envoyer aux autres serveurs en multicast. Le champ client est ajouté pour permettre au dernier serveur d'identifier le destinataire de la réponse. La portée de la case qui a reçu en premier le message permettra au client de corriger son image du fichier. Lorsque le nouveau serveur recevra le message, il saura, par le nom de la fonction, qu'il s'agit d'une redirection. Dans RP*C, il faut au maximum une seule redirection à un message pour arriver sur le bon serveur. Cela est dû au fait que les messages de redirection sont reçus par tous les serveurs.

I. 2-2 Messages d'accusé de réception

RP*N : le client demande explicitement au serveur de lui retourner un accusé de réception.

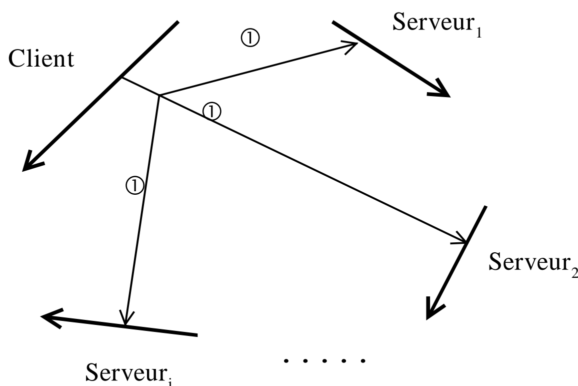
RP*C : l'accusé de réception est automatique pour tous les messages. Il contient une valeur booléenne pour préciser s'il s'agit d'un acquittement ou pas et l'IAM qui contient les portées des cases par lesquelles est passée la requête. En cas de redirection, c'est le dernier serveur qui prend en charge l'envoi du message d'accusé de réception.

II. Protocole de communication RP*N

II. 1 Protocole d'insertion

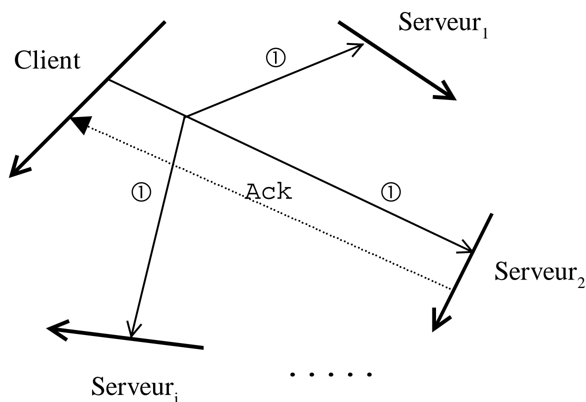
Le client, qui désire insérer un nouvel enregistrement de taille inférieure à 64 Ko dans un fichier, utilise le protocole UDP. Il envoie un message de demande d'insertion (*insert_request*) aux serveurs. Les données à insérer sont encapsulées dans le message. Chaque serveur vérifie si la clé de l'enregistrement appartient à sa portée et, celui qui trouve, exécute l'opération. Un message d'acquittement est envoyé au client s'il en avait fait la demande dans le message d'insertion. Si le serveur n'effectue pas correctement l'insertion pour une raison quelconque, il envoie un message d'erreur au client (cf. figure II.1, figure II.2).

Si la taille des données est supérieure à 64 Ko, le protocole TCP est utilisé. Le client envoie le message *insert_request* aux serveurs avec uniquement la clé de l'enregistrement. Le serveur, qui doit recevoir l'enregistrement, répond au client par un message *insert_reply* pour lui préciser son adresse. Lorsque le client reçoit ce message, il envoie les données au serveur par des messages *send_data*. Le serveur envoie au client un message d'acquittement pour chaque paquet reçu (cf. figure II.3).



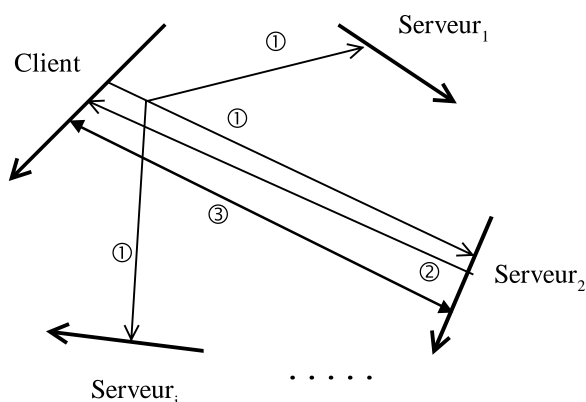
① insert_request()

Figure II.1 : Demande d'insertion (UDP sans acquittement).



① insert_request()
Ack : accusé de réception

Figure II.2 : Demande d'insertion (UDP avec acquittement)

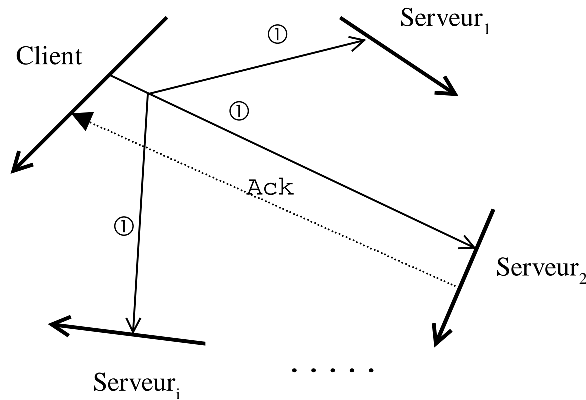


① insert_request()
② insert_reply()
③ send_data()/Ack

Figure II.3 : Demande d'insertion (TCP)

II. 2 Protocole de Suppression

Le client qui désire supprimer un enregistrement envoie une demande de suppression (*delete_request*) en multicast aux serveurs. Le serveur concerné exécute l'opération. On peut envisager de n'envoyer un message au client que si un problème survient (cf. figure II.4).

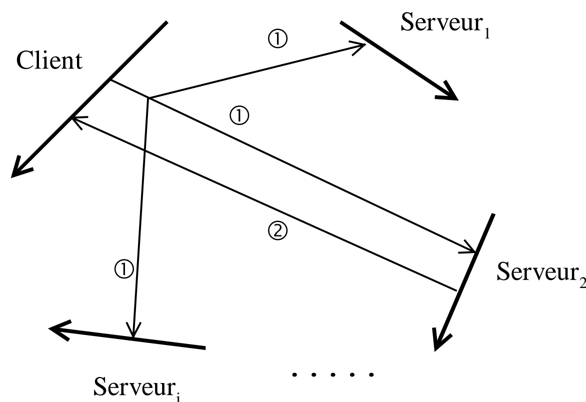


① delete_request()
Ack : accusé de réception

Figure II.4 : Demande de suppression

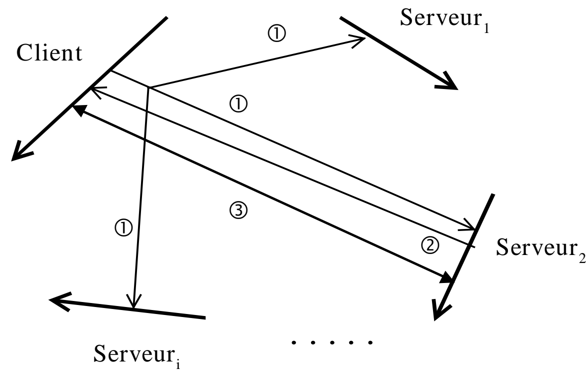
II. 3 Recherche d'un enregistrement

Le client envoie une demande de recherche (*seek_request*) en précisant un port d'écoute et la liste des protocoles qu'il reconnaît. Le serveur concerné devra utiliser l'un de ces protocoles pour envoyer les données. Lorsque la taille des données est inférieure à 64Ko, l'envoi des données peut se faire sans préavis avec le protocole UDP (cf. figure II.5). Si les données sont volumineuses, le serveur utilise la procédure d'envoi des données (canal de données sous TCP). Il envoie alors un message *seek_reply* avec uniquement son adresse en préavis de réponse. Dés réception de ce message, le client ouvre un canal de données et le serveur commence l'envoi effectif du résultat de la recherche par des messages *send_data* (cf. figure II.6).



① seek_request()
② seek_reply()

Figure II.5 : Recherche d'un article (UDP)

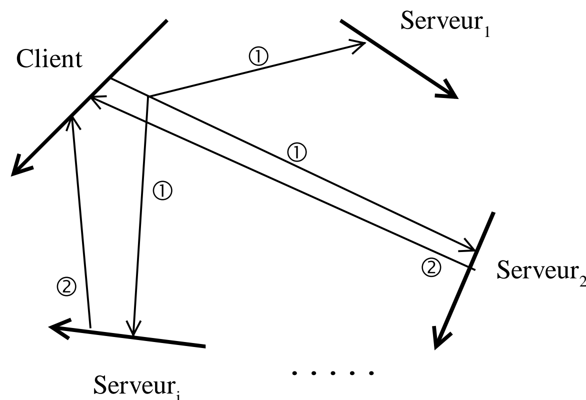


① seek_request()
② seek_reply()
③ send_data()/Ack

Figure II.6 : Recherche d'un article (TCP)

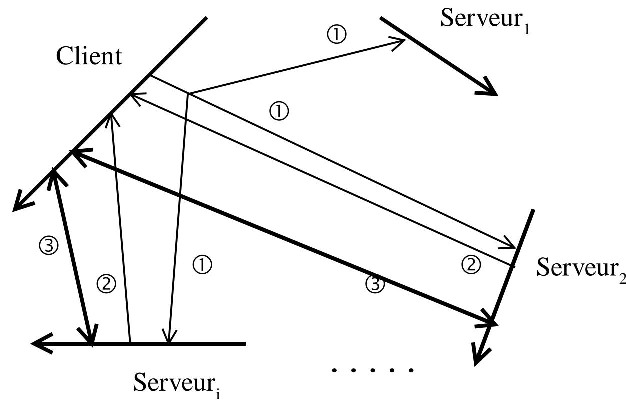
II. 4 Recherche par intervalle

Le client envoie une requête (*seek_range_request*) avec les bornes inférieure et supérieure de l'intervalle de recherche aux serveurs en multicast. Chaque serveur vérifie l'intersection entre sa portée et l'intervalle. S'il existe des enregistrements concernés, il envoie une réponse au client, sinon, il ignore la requête. Si les données ne sont pas volumineuses, la réponse est envoyée en un seul message *seek_range_reply* par UDP (cf. figure II.7). Si les données sont supérieures à 64Ko, la réponse contient alors uniquement l'adresse et la portée du serveur. C'est au client de collecter les réponses et de contacter à nouveau les serveurs suivant l'ordre dans lequel il désire récupérer les résultats. Pour chaque serveur, il ouvrira un canal de données et l'envoi des résultats se fera par des messages *send_data* (cf. figure II.8). A la fin de la requête, l'union des portées des serveurs qui ont envoyé des réponses doit couvrir l'intervalle de recherche.



① seek_range_request()
② seek_range_reply()

Figure II.7 : Recherche par intervalle (UDP)



① seek_range_request()
② seek_range_reply()
③ send_data()/Ack

Figure II.8 : Recherche générale par intervalle (TCP)

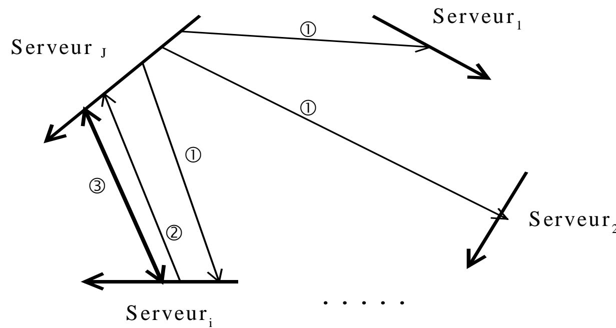
II. 5 Protocole d'éclatement

Un fichier est subdivisé en cases implémentées sur plusieurs sites. Chaque site héberge une seule case. Une case a un nombre maximal d'enregistrements qu'elle peut contenir. Si un serveur reçoit une demande d'insertion alors que ce nombre est déjà atteint, il devra trouver un nouveau site sur lequel va s'effectuer son éclatement. La recherche du nouveau site peut être à la charge du site en débordement ou d'un coordinateur. On distingue alors trois cas : liste des serveurs, site coordinateur d'éclatement et réseau de contrat.

II. 5-1 Liste des serveurs

Chaque serveur dispose d'une liste complète des autres serveurs. Il peut la parcourir pour choisir le premier disponible (qui n'héberge aucune case). Le serveur envoie une demande d'éclatement (*eclat_request_server*) au serveur choisi et attend une confirmation. Un serveur disponible, qui reçoit ce message, répond par un message d'acceptation (*eclat_reply*). Les serveurs, qui hébergent déjà des cases, ne répondent pas aux messages *eclat_request_server*.

Si le serveur ne reçoit pas de message *eclat_reply* jusqu'à l'expiration d'un time-out, alors il envoie un *eclat_request_server* à un autre serveur. Le processus boucle jusqu'à la réception d'un *eclat_reply*. Le transfert des données se fait par des messages *eclat_send_data* et l'acquittement (*eclat_ack_data*) est obligatoire pour chaque paquet afin de permettre au serveur en débordement de supprimer sur son site les données déjà envoyées en toute sécurité (cf. figure II.9). Cette politique d'éclatement est facile à mettre en œuvre mais la liste des serveurs au niveau de chaque site pose des problèmes de mise à jour en cas d'évolution du nombre de serveurs.



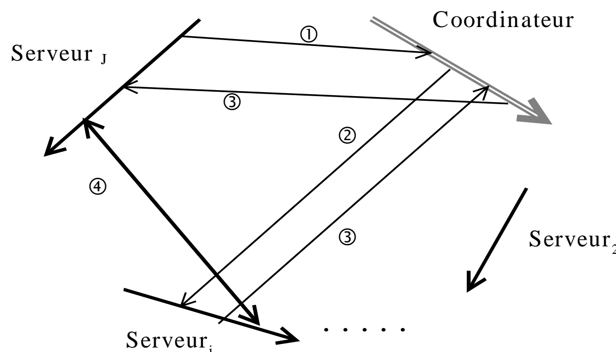
① eclat_request_server()
② eclat_reply()
③ eclat_send_data()/eclat_ack_data()

Figure II.9 : Eclatement d'une case(avec liste des serveurs au niveau de chaque nœud)

II. 5-2 Site coordinateur

Une amélioration de la liste de serveurs au niveau de chaque serveur consiste à disposer cette liste uniquement au niveau d'un site coordinateur d'éclatement qui se chargera de gérer une table d'allocation des serveurs. Donc, un site, qui déborde, s'adresse au coordinateur par un message de demande de nouveau site (*eclat_request_coord*). Apres réception du message *eclat_request_coord*, le coordinateur détermine un site disponible et y envoie un préavis d'éclatement (*eclat_request_server*). Le site désigné répond au coordinateur par un message *eclat_reply*. Le site coordinateur met à jour sa table d'allocation des serveurs et retourne le message *eclat_reply* au site en débordement avec l'adresse du serveur d'accueil. Dés réception de ce message, un canal de données est ouvert et le transfert est effectué par des messages *eclat_send_data* avec acquittement obligatoire par *eclat_ack_data* (cf. figure II.10).

Si le fichier subit beaucoup d'éclatements, le site coordinateur constitue alors un goulot d'étranglement qui détériore les performances du système.



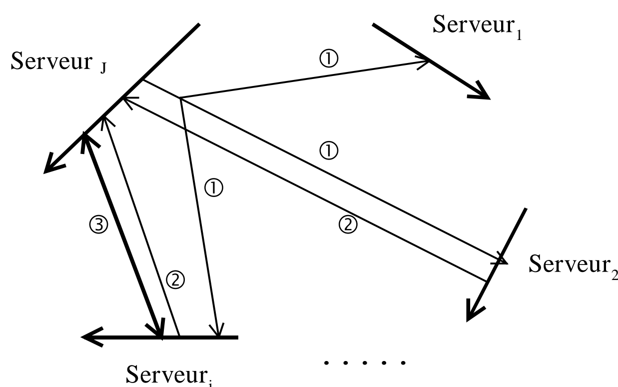
① eclat_request_coord()
② eclat_request_server()
③ eclat_reply()
④ eclat_send_data()/eclat_act_data()

Figure II.10 : Eclatement d'une case avec un site coordinateur

II. 5-3 Réseaux de contrat

Il n'existe pas de liste des serveurs, ni de site coordinateur. Un site en débordement envoie une demande de site disponible (*eclat_request_serveur*) en multicast aux serveurs. Les sites libres répondent en donnant leurs adresses par des messages *eclat_reply*. Le choix est fait sur l'un d'eux et le transfert est effectué par des messages *eclat_send_data* avec acquittement obligatoire (cf. figure II.11).

Le réseau de contrat constitue le meilleur choix comme politique d'éclatement car il rend le système souple en cas d'évolution du nombre de serveurs.



① eclat_request_server()
② eclat_reply()
③ eclat_send_data()/eclat_ack_data()

Figure II.11 : Eclatement d'une case dans un réseau de contrat

III. Protocole de communication RP*C

Chaque client a une image du fichier. Avant d'effectuer une opération, le client calcule l'adresse du serveur concerné. S'il en trouve une, il envoie un message unicast ; à défaut, il envoie un message multicast aux serveurs. L'adresse trouvée peut être erronée si le serveur concerné a subi un éclatement qui n'est pas encore pris en compte dans l'image du client.

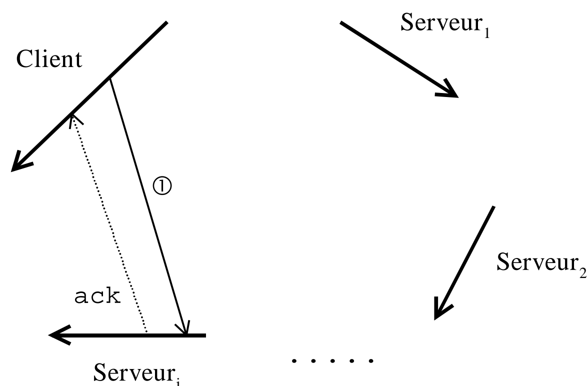
A chaque fois qu'un serveur reçoit un message, il vérifie si la clé contenue dans le message appartient à sa portée. Si c'est le cas, il traite le message et envoie un accusé de réception au client. Si la clé n'appartient pas à sa portée et si le message était unicast, il le redirige en multicast aux autres serveurs. Pour toute requête d'un client, on se retrouve dans l'une de ces trois situations :

- Le client trouve à partir de son image une adresse correcte du serveur ;
- Le client trouve à partir de son image une adresse incorrecte du serveur ;
- Le client ne trouve pas à partir de son image une adresse du serveur.

III. 1 Protocole d'insertion

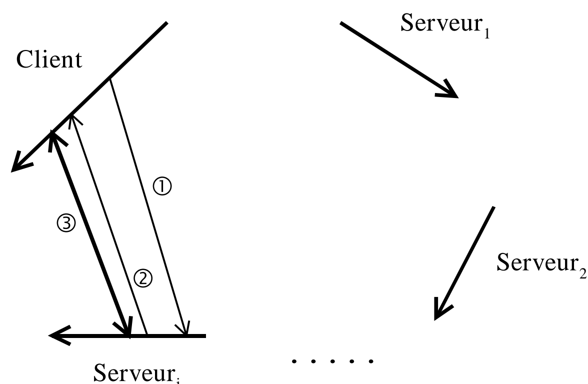
III. 1-1 Insertion avec image correcte

Le client envoie un message de demande d'insertion (*insert_request*) au serveur dont il a trouvé l'adresse. L'enregistrement est encapsulé dans la demande d'insertion si sa taille est inférieure à 64 Ko et le protocole utilisé est UDP. Apres l'insertion, le serveur envoie un acquittement avec sa portée au client (cf. figure III.1). Si la taille de l'enregistrement est supérieure à 64 Ko, la demande d'insertion contient alors uniquement la clé. Le serveur répond au client par un message d'invitation à envoyer les données (*insert_reply*). Le transfert de l'enregistrement du client au serveur se fait sur un canal de données sous TCP par des messages *send_data* (cf. figure III.2).



① insert_request()
Ack : accusé de réception

Figure III.1 : Demande d'insertion (UDP avec image correcte du fichier).

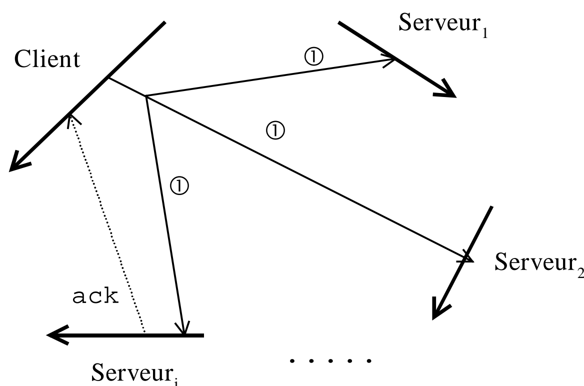


① insert_request()
② insert_reply()
③ send_data()/Ack

Figure III.2 : Demande d'insertion (TCP avec image correcte du fichier).

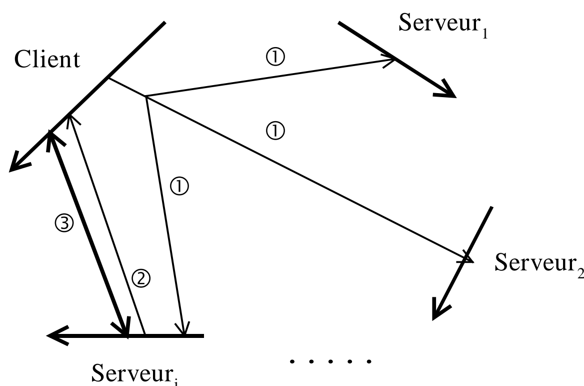
III. 1-2 Insertion sans adresse du serveur

Le client envoie la demande d’insertion (*insert_request*) en multicast aux serveurs. Seul le serveur concerné exécute l’opération et envoie au client un acquittement avec sa portée. Le client choisit le protocole en fonction de la taille des données : UDP (Figure III.3) ou TCP (cf. figure III.4).



① insert_request()
Ack : accusé de réception

Figure III.3 : Demande d'insertion (UDP sans adresse du serveur).

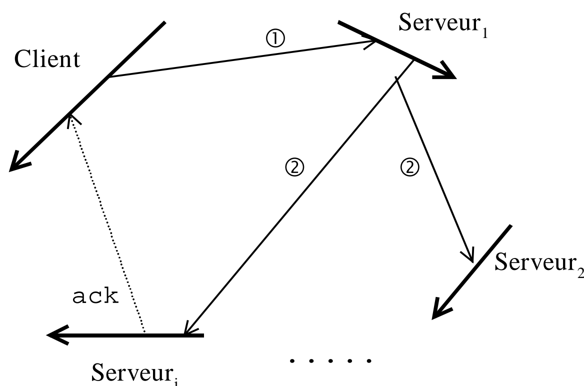


① insert_request()
② insert_reply()
③ send_data()

Figure III.4 : Demande d'insertion (TCP sans adresse du serveur).

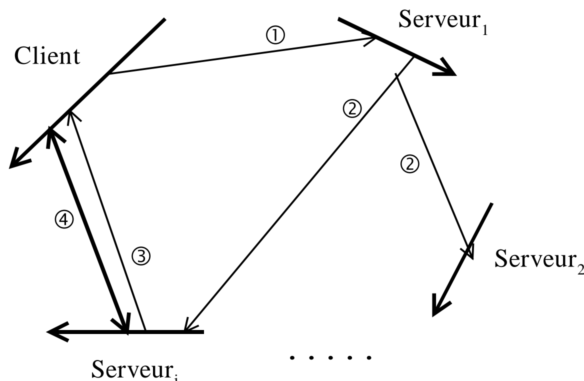
III. 1-3 Insertion avec image fausse

Le client envoie au serveur dont il a trouvé l'adresse une demande d'insertion. Le serveur compare la clé contenue dans le message avec sa portée et constate l'erreur d'adressage. Il redirige alors le message vers les autres serveurs. Le serveur adéquat exécute la requête et envoie un acquittement avec sa portée au client. Le client se servira de cet acquittement pour corriger son image du fichier (cf. figure III.5, figure III.6).



① insert_request()
② insert_request_forward()
Ack : accusé de réception

Figure III.5 : Demande d'insertion (UDP avec image fausse du fichier).



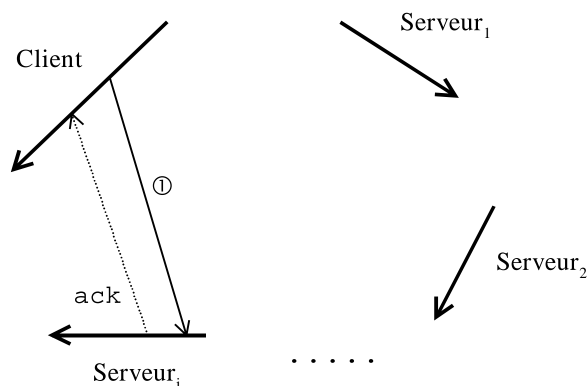
① insert_request()
② insert_request_forward()
③ insert_reply()
③ send_data()/Ack

Figure III.6 : Demande d'insertion (TCP avec image fausse du fichier).

III. 2 Protocole de Suppression

III. 2-1 Suppression avec image correcte

Le client envoie au serveur une demande de suppression d'un enregistrement (*delete_request*). Le serveur vérifie si l'enregistrement dont la clé se trouve dans le message *delete_request* est bien dans sa case. Il supprime ensuite l'enregistrement et retourne un message d'acquittement avec sa portée au client. Si une erreur quelconque se produit, il envoie une notification au client (cf. figure III.7).

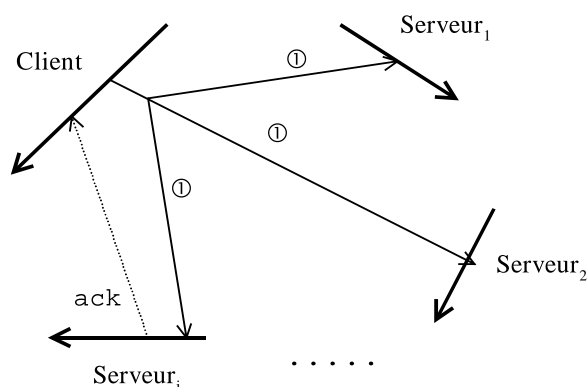


① delete_request()
Ack : accusé de réception

Figure III.7 : Demande de suppression (avec image correcte du fichier).

III. 2-2 Suppression sans adresse du serveur

Le client envoie la demande de suppression en multicast aux serveurs. Le serveur concerné exécute la requête et répond au client par un message d'acquittement ou un message d'erreur (cf. figure III.8).

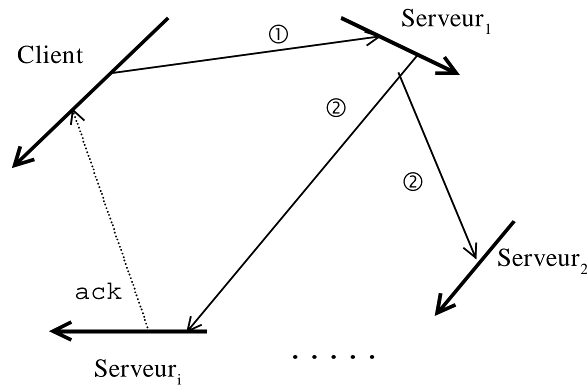


① delete_request()
Ack : accusé de réception

Figure III.8 : Demande de suppression (sans adresse du serveur).

III. 2-3 Suppression avec image fausse

Le client envoie une demande de suppression (*delete_request*) à un serveur avec la clé d'un enregistrement n'appartenant pas à sa portée. Le serveur encapsule sa portée et l'adresse du client dans le message *delete_request* pour former le message de redirection (*delete_request_forward*). Ce message est envoyé vers les autres serveurs. Le serveur adéquat reçoit ce message, traite la requête et envoie une réponse au client (cf. figure III.9).



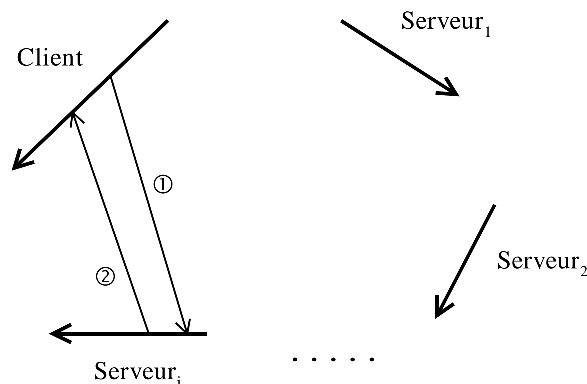
① delete_request()
② delete_request_forward()
Ack : accusé de réception

Figure III.9 : Demande de suppression (avec image fausse du fichier).

III. 3 Recherche d'un enregistrement

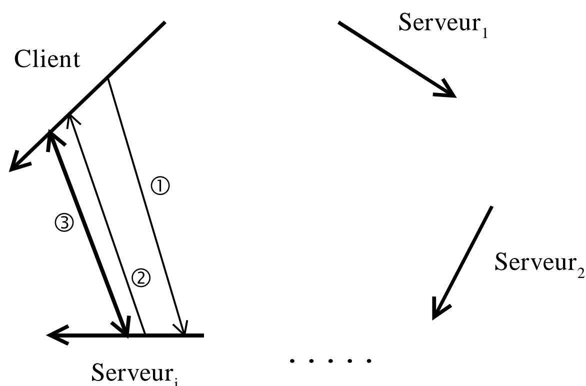
III. 3-1 Recherche avec image correcte

Le client trouve l'adresse correcte du serveur où se trouve l'enregistrement cherché. Il envoie une requête de recherche (*seek_request*) avec la clé de l'enregistrement. Le serveur envoie les données directement dans le message de réponse (*seek_reply*) par UDP si la taille de l'enregistrement est inférieure à 64 Ko (cf. figure III.10). Si la taille de l'enregistrement est supérieure à 64 Ko, alors le message *seek_reply* contient uniquement l'adresse du serveur, l'envoi des données se faisant par un canal de données sous TCP (cf. figure III.11).



① seek_request()
② seek_reply()

Figure III.10 Recherche d'un enregistrement (UDP avec image correcte du fichier).

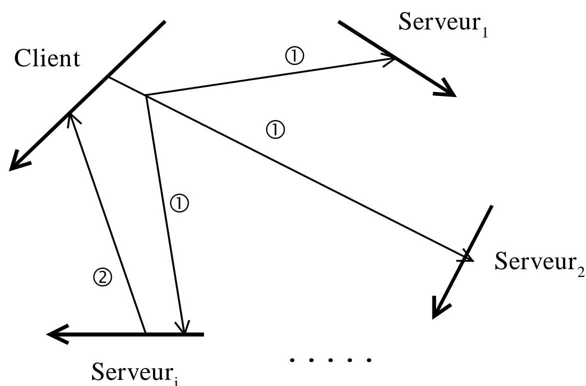


① seek_request()
② seek_reply()
③ send_data()/Ack

Figure III.11 : Recherche d'un enregistrement (TCP avec image correcte du fichier).

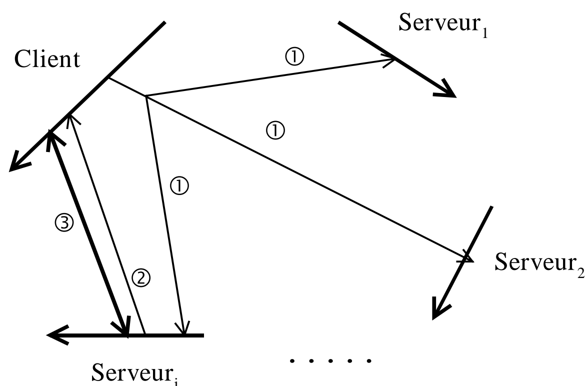
III. 3-2 Recherche sans adresse du serveur

Le message *seek_request* est envoyé en multicast aux serveurs. Le serveur concerné répond directement par le message *seek_reply* avec les données sous UDP (cf. figure III.12) ou bien par un canal de données vers le client sous TCP (cf. figure III.13).



① seek_request()
② seek_reply()

Figure III.12 Recherche d'un enregistrement (UDP sans adresse du serveur).

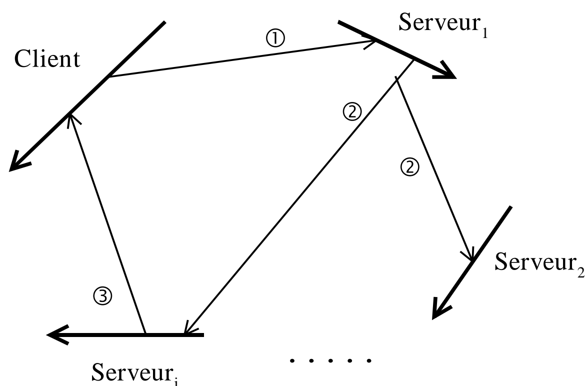


① seek_request()
② seek_reply()
③ send_data()/Ack

Figure III.13 : Recherche d'un enregistrement (TCP sans adresse du serveur).

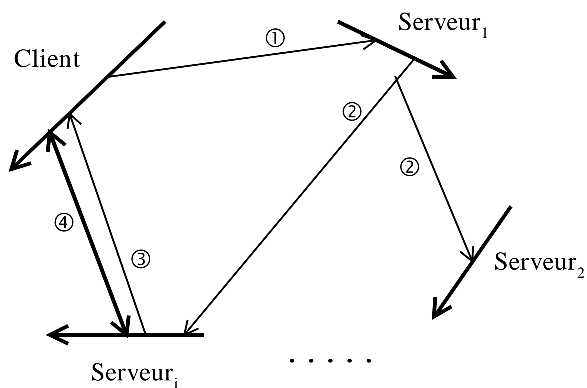
III. 3-3 Recherche avec image fausse

Le client envoie à un serveur le message *seek_request*. Le serveur constate l'erreur d'adressage et envoie un message de redirection (*seek_request_forward*) vers les autres serveurs. Le serveur correct répond par un message *seek_reply* avec les données sous UDP (cf. figure III.14) ou par un canal de données sous TCP (cf. figure III.15)



① insert_request()
② insert_request_forward()
③ seek_reply()

Figure III.14 Recherche d'un enregistrement (UDP avec image fausse du fichier).



① seek_request()
② seek_request_forward()
③ seek_reply()
④ send_data()/Ack

Figure III.15 : Recherche d'un enregistrement (TCP avec image fausse du fichier).

IV. Description des messages

IV. 1 Insertion

Demande d'insertion par le client

insert_request (Id_Cde, algo, file, CB, ack, protocol_list, key, data);

Id_Cde : identifiant de la commande passée par le message : insertion, recherche, suppression, éclatement, etc.

algo : identifiant de l'algorithme : LH*, RP*N, RP*C, RP*S…

file : identifiant du fichier où les données seront insérées.

key : clé de l'article à insérer.

CB : Control Block. Contrôle d'accès, droits d'accès...

ack : pour la version 0, il est toujours à zéro ; c'est-à-dire pas d'acquittement.

protocol_list : liste des protocoles reconnus et acceptés par le client (TCP, UDP).

Data : données à insérer.

Redirection du message

insert_request_forward (Id_Cde, algo, file, client, CB, IAM, ack, protocol_list, key, data);

La plupart des paramètres correspondent à ceux du message insert_request. Seuls les champs Client et IAM ont été ajoutés.

Client : Adresse du client qui a initié la requête

L'IAM (Image Adjustment Message) : portée de la case SDDS du serveur qui envoie le message de redirection (ce champ n'est pas utilisé par l'algorithme RP*N)

Réponse du serveur où l'article doit être inséré

insert_reply (Id_Cde, algo, file, CB, forward, IAM, ack, data_port, data_protocol, key);

forward : une valeur TRUE indiquera au client que son message insert_request a été redirigé et qu'il doit ainsi prendre en compte la valeur du message IAM.

IAM : Image Adjustement Message. Portée de la case SDDS du serveur où le client a envoyé la clé.

ack : Zéro s'il n'y a pas d'acquittement, un s'il y'a acquittement.

data_port : port par lequel devront arriver les données du client.

data_protocol : protocole choisi par le serveur dans la liste *protocol_list* envoyée par le client. C'est ce protocole que le client devra utiliser pour envoyer les données.

Envoi des données par le client

send_data (Id_Cde, algo, file, ack, key, data) ;

Ce message utilise le protocole spécifié dans la réponse (*insert_reply*) du serveur qui va recevoir les données.

IV. 2 Suppression

Demande de suppression d'un article

delete_request (Id_Cde, algo, file, key, time-out) ;

Redirection de la demande de suppression d'un article

delete_request_forward (Id_Cde, algo, file, key, Client, IAM, time-out) ;

Réponse du serveur

Delete_ack (Id_Cde, algo, file, key, CB) ;

IV. 3 Recherche d'un enregistrement

Demande de recherche d'une clé

seek_request (Id_Cde, algo, file, key, CB, data_port, protocol_list)

Redirection d'une demande de recherche d'une clé

seek_request_forward (Id_Cde, algo, file, key, client, CB, data_port, protocol_list, IAM)

Réponse du serveur - Message de données.

seek_reply (Id_Cde, algo, file, key, CB, data) ;

IV. 4 Recherche par intervalle

Requête multicast

seek_range_request(Id_Cde, algo, file, servers_multicast, key_min, key_max, data_port, data_protocol, time-out)

key_min : borne inférieure de l'intervalle de recherche

key_max : borne supérieure

data_port : port par lequel devront arriver les résultats.

servers_multicast : adresse du groupe de serveurs qui recevront le message.

Réponse d'un serveur

seek_range_reply (Id_Cde, algo, file, key_min, key_max, CB, data) ;

IV. 5 Eclatement

Demande d'éclatement de serveur i à serveur j

eclat_request_server (Id_Cde, algo, file, level, size_bucket, CB, protocol_list, time-out);
level : portée de la case en débordement
size_bucket : taille de la case à éclater dans le serveur i.
protocol_list : liste des protocoles reconnus et acceptés par le serveur i.
time-out : le serveur i attend la réponse du serveur j jusqu'à l'expiration de ce time-out.

Demande d'éclatement de serveur i au coordinateur

eclat_request_coord (Id_Cde, algo, file, level, size_bucket, CB, protocol_list, time-out);

Réponse du serveur récepteur à une demande d'éclatement

eclat_reply (Id_Cde, algo, file, CB, data_port, data_protocol, time-out);

Envoi des données par le serveur en débordement

eclat_send_data (Id_Cde, algo, file, data) ;

Compte-rendu du serveur récepteur au serveur émetteur

eclat_ack_data (Id_Cde, algo, file, CB) ;

Chapitre 3
Outils techniques pour la mise en œuvre du prototype

I. Interface de communication entre processus distants

Le modèle des sockets a été initialement introduit dans Berkeley UNIX® (BSD) au début des années 1980. Les sockets ont été conçues comme des mécanismes de communication inter-processus (IPC) local. Plus tard, ils ont évolué vers des mécanismes de communication inter-processus réseau [SINHA96].

Une socket est définie comme étant un point terminal bidirectionnel pour la communication entre processus ; c'est-à-dire permettre la transmission et la réception de données à travers ces connexions. Le modèle des sockets Windows offre un service pour les protocoles avec connexion (socket séquence-de-données (stream)) et sans connexion (socket datagramme).

En première approximation, les modèles de communication qui sont accessibles à travers les sockets sont tout à fait analogues à deux outils de la vie courante : le courrier et le téléphone. Dans tous les cas de communication, il y a au moins deux entités qui communiquent : il faut donc au moins deux points d'entrée.

Le courrier

L'analogie de la socket est ici une boîte aux lettres. Le message est une lettre qui porte une adresse. L'expéditeur dépose la lettre dans la boîte aux lettres du service postal. L'adresse permet aux services postaux de déposer le message dans la boîte aux lettres du destinataire.

Le téléphone

Dans ce modèle de communication, les deux entités établissent une connexion directe entre leurs postes téléphoniques respectifs. Le numéro du poste de l'appelé doit être connu pour que l'initiateur de la connexion puisse le composer. Le modèle de messages échangés est un flot bidirectionnel : il n'y a pas de notion de frontières entre des messages. L'analogie de la socket est ici un poste téléphonique.

I. 1 Caractéristiques d'une socket

I. 1-1 Propriétés d'une communication

a) Fiabilité

b) Préservation de l'ordre des données

c) Non-duplication des données

d) Communication en mode connecté

e) Préservation des limites de messages

I. 1-2 Le domaine d'une socket

Une socket peut être utilisée sur plusieurs familles d'adresses ou domaines. Chaque famille d'adresses a sa propre syntaxe de représentation des adresses réseaux qui dépendent des protocoles utilisés. On distingue principalement les domaines suivants :

- AF_INET : domaine Internet protocole TCP/IP ;
- AF_IPX : protocoles IPX, SPX ;
- AF_NETBIOS : protocoles NetBeui.

I. 1-3 Les types de sockets disponibles

Le type SOCK_DGRAM :

Mode non connecté, envoi de datagrammes de taille bornée, préservation des limites de messages. Dans le domaine Internet, le protocole sous-jacent est UDP.

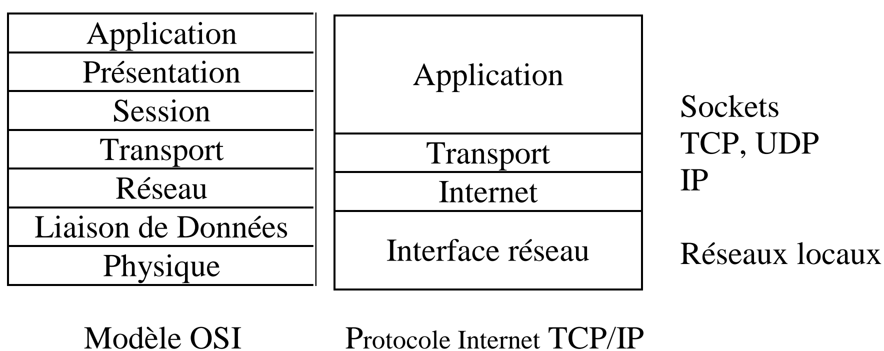
Le type SOCK_STREAM

Communications fiables en mode connecté. Dans le domaine Internet, le protocole sous-jacent est TCP.

I. 1-4 La famille de protocoles TCP/IP

Les Protocoles TCP/IP correspondent à un modèle composé des quatre couches suivantes : Application, Transport, Internet et Interface réseau.

Ce modèle est souvent nommé famille de protocoles TCP/IP. Chaque couche du modèle TCP/IP correspond à une ou plusieurs couches du modèle OSI (Open Systems Interconnection) défini par l'ISO (International Standards Organization).



La manière dont les ordinateurs sont connectés et communiquent dépend des protocoles définis à l'intérieur des quatre couches de TCP/IP. Les protocoles les plus utilisés sont TCP (Transmission Control Protocol), UDP (User Datagram Protocol), IP (Internet Protocol).

Protocole TCP : Protocole de contrôle de transmission

Il offre un service fiable de remise de paquets, orienté connexion, au-dessus du protocole IP. TCP garantit la remise des paquets et l'ordre approprié des données ainsi qu'il fournit une somme de contrôle qui vérifie l'intégrité de l'en-tête des paquets et des données qu'ils contiennent. Si un paquet TCP est endommagé ou perdu par le réseau au cours de la transmission, TCP retransmet ce paquet. Cette fiabilité fait de TCP un protocole bien adapté pour la transmission de données lors d'une session, pour les applications Client-Serveur et les services importants.

Cette fiabilité engendre des contraintes : en effet, des bits supplémentaires sont nécessaires afin de remettre les informations dans l'ordre approprié ainsi qu'une somme de contrôle (checksum) obligatoire pour garantir la fiabilité des paquets TCP. Afin d’assurer la bonne remise des données, le destinataire doit émettre un accusé de réception.

Ces accusés de réception (ACK) génèrent un supplément de trafic sur le réseau et ralentissent le débit de transmission des données mais en faveur d'une plus grande fiabilité.

Protocole UDP : Protocole de datagramme utilisateur

Si la fiabilité n'est pas un critère essentiel, UDP un complément du protocole TCP, offre un service de datagrammes au-dessus du protocole IP. Ce service est sans connexion et ne garantit ni la remise, ni l'ordre d'arrivée des paquets délivrés. Ceci permet d'échanger des

données sur des réseaux à fiabilité élevée sans utiliser inutilement des ressources réseau ou du temps de traitement.

Le protocole UDP prend également en charge l'envoi de données d'un unique expéditeur vers plusieurs destinataires en même temps. Cette diffusion de groupe, appelée diffusion multipoint ou multicast, est un outil important pour la mise en œuvre de système Client-Serveur où un client désire s'adresser à un groupe de serveurs notamment dans les SDDS.

Protocole IP : Protocole Internet

Le protocole IP permet la remise des paquets pour tous les autres protocoles de la famille TCP/IP. Il fournit un système de remise de données optimisé, sans connexion. Il n'existe aucune garantie quant à la remise des paquets IP à leur destinataire ni leur réception dans l'ordre dans lequel ils ont été envoyés. Ainsi donc, seuls les protocoles de niveau supérieur sont responsables des données contenues dans les paquets IP et de leur ordre.

I. 1-5 Création, identification et destruction d'une socket

La création d'une socket se fait grâce à l'appel de la fonction *socket().* Cette fonction rend un entier qui servira à identifier la socket créée. Il s'agit donc d'une ressource qui est du même genre que celle qui est allouée par la fonction d'ouverture de fichier *open*(). De ce fait, la destruction de la socket se fait avec le même appel que celui de la fermeture d'un fichier : *close().*

Une socket a trois composantes primaires :

L'interface à laquelle elle est liée ou attachée (spécifiée par une adresse IP) ;

Le numéro de port (ou ID) auquel les données sont envoyées ou reçues ;

Le type de la socket (séquence-de-données (stream) ou datagramme).

I. 1-6 Attachement d'une socket à une adresse

Pour qu'une socket soit une entité visible sur un réseau, il faut lui associer l'adresse réseau de la machine sur laquelle elle est créée. L'attachement d'une adresse à une socket se fait grâce à l'appel de la fonction *bind*().

Du côté du récepteur, il faut obligatoirement appeler la fonction *bind*() pour que d'autres processus puissent le contacter à travers la socket ouverte. Cette fonction attend

comme argument l'adresse de la machine réceptrice. Il faut aussi y spécifier un numéro de port dans le deuxième champ. Comme les sockets distinguent les communications en utilisant les numéros de port sur une machine, il devient possible de servir en même temps différentes applications serveurs. En affectant un numéro de port différent pour chaque application, on reconnaît les paquets appartenant à chacune d’elles.

I. 2 Communication par datagrammes

I. 2-1 Principe général

Les communications par datagrammes permettent d'échanger des messages analogues au courrier postal :

Les frontières entre les messages sont conservées ;

Il n'y a pas de connexion entre les deux entités communiquantes ;

Chaque message doit contenir l'adresse de son destinataire.

Il n'y a pas de garantie autre que la qualité du contenu du message. Cela signifie que :

L’expéditeur n'est pas assuré de la délivrance de son message : la fiabilité est plus ou moins bonne dans un réseau local (LAN) mais peut être très médiocre dans les WAN tel que le réseau mondial Internet ;

Il ne peut pas avoir d'information sur l'arrivée de son message ;

Un même message peut être délivré plusieurs fois ;

Deux messages émis dans un certain ordre n'arrivent pas forcément dans le même ordre.

Un processus, après avoir ouvert une socket, peut l'utiliser pour communiquer avec plusieurs partenaires, aussi bien en émission qu'en réception.

I. 2-2 Exemple d'échange de données en mode datagramme

Côté émetteur		**Côté récepteur**	
Créer une socket	*socket()*	Créer une socket	*socket()*
-	-	Attacher la socket à une adresse	*bind()*
Envoyer un message	*sendto()*	Recevoir le message	*recvfrom()*

Côté Emetteur

Une fois la socket créée, il suffit de l'utiliser avec la fonction *sendto*().

Une application qui veut envoyer des données doit spécifier l'adresse à laquelle les données sont destinées dans la fonction *sendto().* Cette fonction retourne le nombre d'octets effectivement transmis et -1 en cas d'échec.

Toutefois, les conditions d'échec ne sont que locales, en particulier, si on envoie un message à une machine et un numéro de port où aucun processus n'écoute, il n'y aura pas de compte rendu d'erreur, et le message sera perdu sans que l'émetteur en soit avisé. Par contre, la validité du descripteur de socket et la validité de l'adresse sont vérifiées.

Côté récepteur

Après l'ouverture de la socket et son attachement, le récepteur attend des messages grâce à la fonction *recvfrom*().

Pour pouvoir récupérer un message avec cette fonction, elle doit être appelée après l'attachement de la socket, et après qu'un émetteur a envoyé un message à l'issue de l'opération d'attachement. Cette fonction retourne le nombre d'octets reçus, et -1 en cas d'erreur. C'est une fonction bloquante si aucun caractère n'est disponible pour la lecture. Le processus est réveillé dès qu'un caractère est disponible. Le récepteur peut connaître l'adresse de l'émetteur grâce à l'argument *from* retourné par la fonction *recvfrom*().

I. 3 Communication en mode connecté

I. 3-1 Principe général

Dans le domaine Internet, ce mode de communication s'appuie sur le protocole TCP. Il accroît donc le volume d'octets transférés sur le réseau mais apporte en contrepartie la fiabilité de la communication.

Avant la connexion, l'une des deux entités doit attendre une connexion, et l'autre la demander. Une fois la connexion établie, les deux entités jouent un rôle symétrique : les rôles d'émetteur et de récepteur sont confondus.

Lorsqu'une connexion est établie avec une socket séquence-de-données (stream), un circuit virtuel est établi entre les deux entités communiquant sur les deux machines. Ce circuit reste ouvert jusqu'à ce que les deux applications décident d'arrêter l'envoi de données sur le

circuit (en appelant la fonction *close()*) ou jusqu'à ce qu'il y ait une erreur qui provoque la fin anormale du circuit de communication.

Outre la fiabilité apportée par la communication avec le protocole TCP, un autre aspect de ce mode de communication est l'aspect continu de l'information : il n'y a pas de frontières entre les messages.

I. 3-2 Exemple d'échange de données en mode connecté

Côté émetteur		**Côté récepteur**	
Créer une socket	*socket()*	Créer une socket	*socket()*
-	-	Attacher la socket à une adresse	*bind()*
-	-	Ecouter les demandes de connexions	*listen()*
Demander une connexion	*connect()*	-	-
-	-	Accepter la connexion	*accept()*
Envoyer des données	*send()*	Recevoir les données	*recv()*

I. 3-3 L'attente et l'établissement de la connexion

Une première socket d'écoute est ouverte par le récepteur, et cette socket est mise en attente de connexion avec la fonction *listen*(). Un paramètre important de la fonction *listen*() est la taille de la file des demandes de connexions.

Lorsqu'une demande de connexion arrive sur cette socket, l'appel de *listen*() est débloqué, le processus d'écoute réveillé, et une nouvelle socket est créée avec l'appel de la fonction *accept*(). C'est cette deuxième socket qui est connectée à l'émetteur.

Pour réaliser un véritable serveur, il faut créer un nouveau processus dès qu'une demande de connexion arrive, laisser le processus fils appeler la fonction *accept*(), et traiter le dialogue avec le client, pendant que le père attendra de nouvelles demandes de connexion sur la première socket qu'il a créée. Si on ne procède pas de cette façon, le serveur ne peut alors traiter qu'une seule demande à la fois, et toutes les autres demandes provenant d'autres clients sont mises en attente jusqu'à ce que le serveur ait terminé la transaction avec le client en cours de traitement.

I. 3-4 La demande de connexion par le client

La demande de connexion par le client est réalisée par l'appel de la fonction *connect*(). Cette fonction réussit à condition qu'il y ait bien un processus en attente de connexion (fonction *listen*()) sur la machine et le port désigné par l'adresse passée à la fonction *connect*() et que la file des demandes de connexions ne soit pas pleine.

I. 3-5 Le dialogue serveur/client

L'envoi de données sur la connexion établie par *connect*() et *accept*(), est bidirectionnel. Chacune des deux entités peut alors envoyer des octets dans ce tuyau. L'ordre des octets est conservé.

Une application envoie les données en utilisant la fonction *send().* Cette fonction nécessite : le descripteur du socket, un pointeur sur le buffer à envoyer, la longueur du buffer et un entier qui spécifie des drapeaux pouvant modifier le comportement de *send().* Les appels de *send()* sont bloquants lorsque le tampon de réception de la socket distante et le tampon d'émission de la socket locale sont pleins.

Pour recevoir les données, une application utilise la fonction *recv()* qui nécessite un pointeur au buffer à remplir avec les données, la longueur de ce buffer et des drapeaux entiers. La fonction *recv()* renvoie le nombre d'octets réellement reçu et *send()* renvoie le nombre d'octets réellement envoyé. Notons que les applications client et serveur doivent toujours vérifier les codes que *send()* et *recv()* retournent pour s'assurer qu'ils ne sont pas différents du nombre attendu.

Cependant, un appel *send()* d'un côté ne correspond pas nécessairement à un appel *recv()* de l'autre côté. C'est-à-dire qu'il peut y avoir fragmentation ou assemblage par le destinataire des blocs de données envoyés par l'émetteur.

Grâce à la nature orientée flux de données des sockets TCP, il n'est pas nécessaire de faire des correspondances un à un entre les deux appels *send()* et *recv().* Par exemple, une application client peut appeler dix fois la fonction *send(),* chacune de 100 octets. Le système peut combiner ces envois en un seul paquet réseau de telle façon que si l'application serveur fait un appel *recv()* avec un buffer de 1000 octets, elle reçoit toutes les données en une seule fois. Donc, une application ne doit faire aucune hypothèse concernant l'arrivée des données. Un serveur attendant 1000 octets doit appeler la fonction *recv()* dans une boucle jusqu'à ce qu'il reçoive toutes les données.

II. La transmission multipoint(Le multicast)

La notion de transmission de données en mode point-à-point (terminal-ordinateur, client-serveur) a évolué pour faire face à de nouveaux besoins : la communication de un vers plusieurs ou plusieurs vers plusieurs.

Cette notion de transmission multipoint (multicast) vise à envoyer un message vers plusieurs sites en évitant de faire circuler de multiples exemplaires des même données sur un même lien, consommant ainsi de la bande passante (cf. Figure 1). Des extensions ont donc été apportées au niveau de la couche IP, de nouveaux protocoles de routage ont été développés et un réseau expérimental de diffusion multipoint se déploie : le MBone.

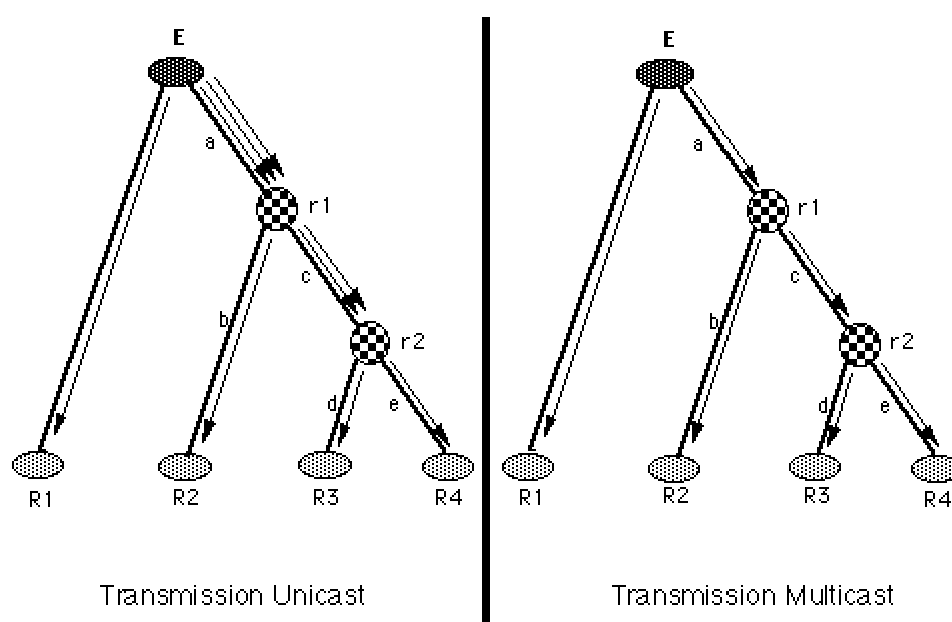


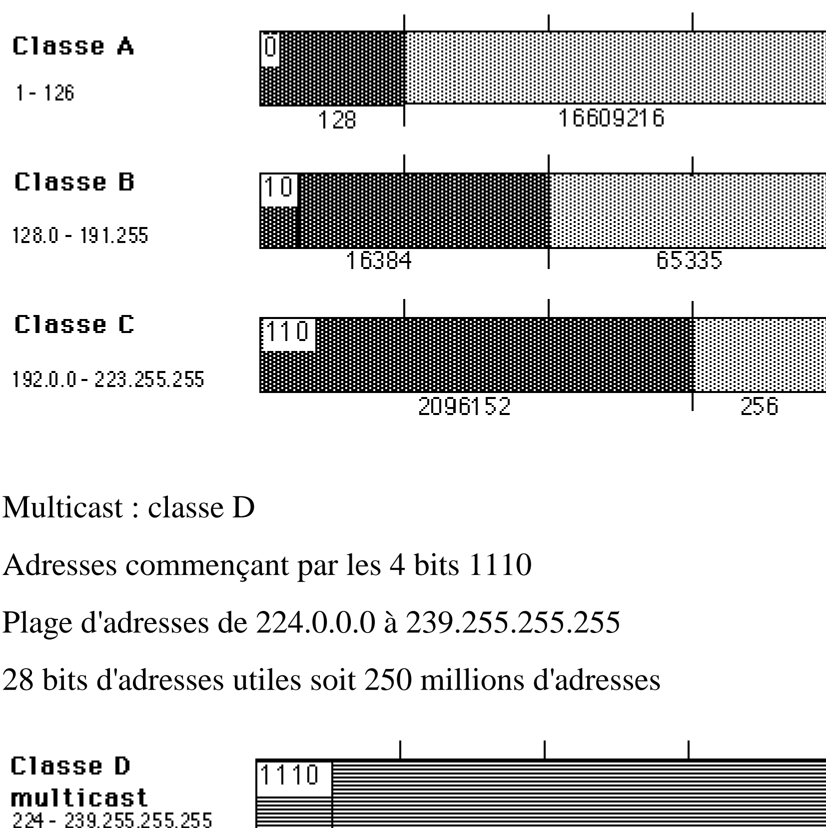
Figure 1 Transmission unicast & multicast

II. 1 Implémentation du multipoint IP

Aux trois classes d'adressage IP traditionnelles (A, B et C) s'ajoute la classe D dite multipoint. Toute adresse IP commençant par 1110 appartient à cette classe et représente une adresse de groupe (cf. Figure 4). Un tel groupe représente un nombre quelconque (y compris nul) d'équipements sur le réseau. En notation habituelle, les adresses de groupe vont de 224.0.0.0 à 239.255.255.255. Un certain nombre d'adresses sont réservées, parmi lesquelles 224.0.0.0 qui ne doit pas être utilisée et 224.0.0.1 qui représente l'ensemble des machines IP

sur le réseau local. Les groupes évoluant dynamiquement, les systèmes rejoignent et quittent à la demande un nombre quelconque de groupes. A noter qu'il n'est pas nécessaire d'être membre d'un groupe pour émettre des données vers ce groupe.

- Classe A : 127 réseaux, 16 millions hôtes/réseau
- Classe B : 16384 réseaux, 65534 hôtes/réseau
- Classe C : 2 millions de réseaux, 254 hôtes/réseau



- Multicast : classe D
- Adresses commençant par les 4 bits 1110
- Plage d'adresses de 224.0.0.0 à 239.255.255.255
- 28 bits d'adresses utiles soit 250 millions d'adresses

Figure 4 : Adressage de groupe

II. 2 Le routage multipoint

Des protocoles de routage spécifiques ont été développés ou sont en cours de spécification pour résoudre les problèmes suivants :

- Comment atteindre les membres des différents groupes répartis sur tout l'Internet (construction d'arbres d'acheminement) ?
- Comment économiser de la bande passante en n'acheminant les paquets multipoint que là où il y a des membres des groupes correspondants ?
- Comment optimiser les échanges entre routeurs (vaut-il mieux annoncer quels sont les groupes que l'on souhaite recevoir ou ceux que l'on ne veut pas recevoir ?)

On peut actuellement répartir les protocoles de routage multipoint en deux familles :

1. Ceux orientés forte densité de clients (dense-mode) : Ces protocoles supposent qu'il y a des membres des groupes multipoint sur la plupart des réseaux et que l'absence de membre constitue l'exception pour laquelle il y aura transfert d'information entre routeurs.
2. Ceux orientés faible densité de clients (sparse-mode) : Ces protocoles supposent, au contraire des précédents, que les membres de groupe multipoint sont très dispersés et peu nombreux par rapport au nombre de réseaux desservis.

II. 3 Protocoles multicast

- IGMP (Internet Group Management Protocol)
  - Les machines déclarent leur appartenance à un ou plusieurs groupes auprès du routeur multipoint dont elles dépendent. Celui-ci diffusera alors les datagrammes destinés à ce ou ces groupes.
  - Le groupe est dynamique : de 0 à une quasi-infinité de membres.
  - Les membres d'un groupe sont indépendants d'une localisation physique.
  - ICMP, fait partie de IP (protocole numéro 2) et comprend essentiellement deux types de messages : un message d'interrogation (Host Membership Query), utilisé par les routeurs, pour découvrir et/ou suivre l'existence de membres d'un groupe et un message de réponse (Host Membership Report), délivré en réponse au premier, par au moins un membre du groupe concerné.
- DVMRP (Distance Vector Multicast Routing Protocol)

  Echange des informations de routage entre routeurs voisins (inspiré de RIP).
- PIM (Protocol Independent Multicast)

  Protocole développé par CISCO, modes dense (DM) et clairsemé (SM).
- RTP (Real-Time Transport Protocol)

  Séquencement des paquets par date (technique de *timestamping*).
- RSVP (Reservation Protocol)

  Réservation des ressources nécessaires à un flot multicast.

II. 4 Supports requis

Le multicasting IP n'est actuellement supporté que :

- Sur des machines dont le driver d'interface a été modifié pour supporter le multicast (protocole IGMP, Internet Group Management Protocol).
- Sur des sockets de famille de protocole `AF_INET` de type `SOCK_DGRAM`, c'est-à-dire en mode datagramme.

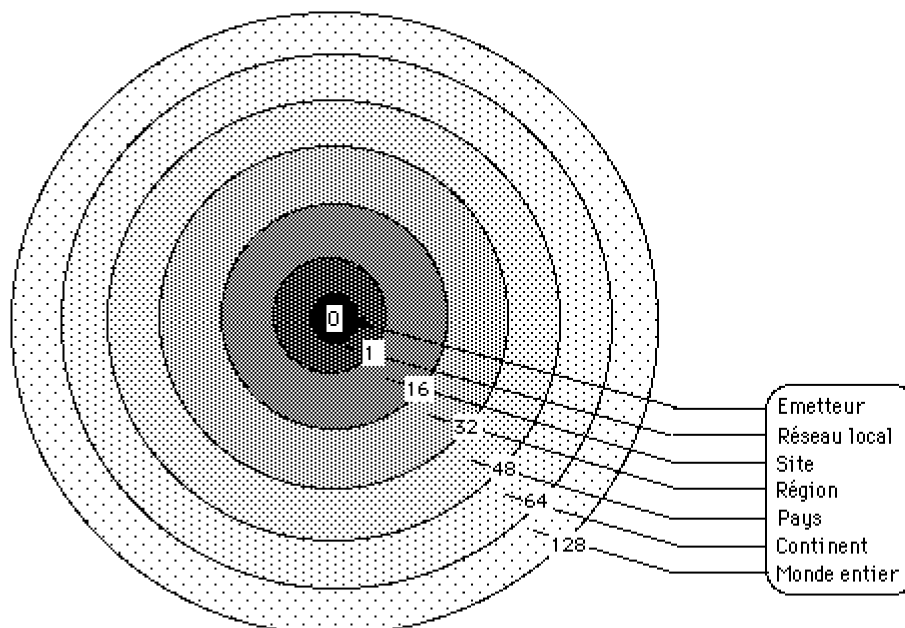
II. 5 Multipoint sur réseau Ethernet

L'adressage multipoint étant défini au niveau de la couche réseau, il reste à la couche MAC à délivrer les trames. Dans le cas d'Ethernet, la notion d'adressage de groupe existe depuis sa spécification dans l'avis IEEE802.3.

II. 6 Emission de datagrammes multicast

Par défaut, les datagrammes multicast sont envoyés avec un TTL (time-to-live) de 1 qui correspond au sous-réseau sur lequel est physiquement connectée la machine par son interface Ethernet. Le TTL contrôle la portée de l'émission (cf. figure 2). Habituellement, un TTL de 15 permet de diffuser les datagrammes sur le site entier, un TTL de 47 permet une diffusion nationale et un TTL de 127 et, au-delà, une diffusion mondiale.

Figure 2 : Portée diffusion : TTL (time to live) : LAN, site, région, continent, monde



II. 7 Réception d'un datagramme multicast

Avant qu'une machine puisse recevoir des datagrammes multicast, elle doit être membre du groupe. L'application peut demander à la machine sur laquelle elle s'exécute de joindre le groupe.

Si une machine envoie un datagramme vers un groupe dont elle est membre, une copie interne au niveau de la couche IP est réalisée pour qu'il soit délivré localement. Une option permet de contrôler ce fonctionnement pour déterminer si oui ou non l'application émettrice doit recevoir le datagramme en écho.

II. 8 Exemple de réseau multicast : Le MBone

MBone, pour Multicast backBone, est le réseau virtuel de diffusion multipoint. Il est issu d'expérimentations de l'IETF, à San Diego, en 1992, visant à diffuser sur l'Internet le son, puis la vidéo. Outre les réunions annuelles de l'IETF, on y trouve de plus en plus de séminaires ou de conférences, des images de satellites météorologiques en quasi-temps réel, des démonstrations lors de salons... Le MBone est également le support de diffusions à moins grande échelle : télé-réunions de travail, tableaux blancs partagés… Autrement dit, les applications de travail coopératif.

Ce réseau est constitué, pour l'instant, de machines exécutant Mrouted qui tissent une toile d'araignée constituée de tunnels au-dessus de l'Internet. Cette situation est transitoire en attendant le support du multipoint dans les routeurs de l'Internet et le choix d'un protocole de routage approprié.

III. Programmation multitâche

III. 1 Processus et Threads

Lorsqu'une application Win32 est activée, un processus est crée. Il s'agit d'une entité statique qui possède des éléments de l'application : code, données statiques et ressources diverses (telles que les fichiers ouverts ou les canaux). Mais, un processus n'existe jamais seul. En même temps que lui, se crée un thread. C'est la partie dynamique de l'application, qui participe au mécanisme du multitâche de Windows NT. En fait, l'Ordonnanceur (la partie du système chargée de la mise en œuvre du multitâche) ne connaît que des threads et ignore les processus. Le premier thread d'une application, qui apparaît en même temps que le processus, s'appelle le thread principal. Une application peut créer d'autres threads de façon dynamique. Un thread dispose d'une pile spécifique et il est caractérisé par la valeur des registres du processeur qui indiquent son état d'exécution [CUSTE93].

Pour créer un nouveau thread, il suffit d'appeler la fonction *CreateThread()*, dont le principal paramètre est l'adresse du point d'entrée du thread. Cette fonction est située dans l'application ou dans un DLL. Juste après l'emploi de *CreateThread()*, l'application s'exécute en deux endroits différents : le premier thread continue de fonctionner à l'instruction qui suit l'appel ; le nouveau thread démarre à la fonction passée en paramètre lors de l'appel. Tous les deux s'exécutent alors parallèlement.

Si la machine sur laquelle fonctionne l'application ne dispose que d'un processeur, les deux threads s'exécutent alternativement. Le système peut, à tout instant, interrompre un thread pour passer la main à un autre. Si le système est multiprocesseur, les deux threads sont susceptibles de fonctionner simultanément, chacun sur un processeur. Tout cela est transparent pour les applications.

Le second thread poursuit son exécution jusqu'à ce qu'on appelle la fonction *ExitThread()* ou qu'il sorte simplement lui-même de sa fonction d'entrée. Dans une application, le thread principal peut contenir les fonctions d'interface avec l'utilisateur (menu, boites de dialogue,...). Les threads créés dynamiquement peuvent servir à réaliser des tâches de fond.

III. 2 Synchronisation de l'Exécution des Thread

Il est nécessaire de synchroniser l'exécution des threads, c'est-à-dire disposer de mécanismes permettant de contrôler leur exécution.

Considérons deux threads d'un même processus fonctionnant simultanément. Le premier écrit des données dans une structure globale de l'application, tandis que le second les lit. Puisqu'ils font partie du même processus, ils peuvent tous deux accéder à cette structure. Si, pendant que le premier thread écrit dans la structure, l'ordonnanceur effectue une commutation de contexte pour passer la main au second, celui-ci peut lire une structure dont les données sont incohérentes puisqu'en partie modifiées. Il faut mettre en place un blocage de la ressource partagée qu'est la structure globale. Le problème est le même pour des threads de processus différents, lorsque la ressource est un fichier, un périphérique ou une zone de mémoire partagée.

Windows NT propose quatre mécanismes de synchronisation : Section critique, Mutex, Sémaphore et Event. La section critique ne peut être utilisée qu'à l'intérieur d'un même processus. Les trois autres permettent la synchronisation de threads situés dans des processus différents.

- La section critique assure l'exclusion mutuelle entre les threads d'un même processus. Elle assure la cohérence des données globales d'un processus face aux accès concurrents des threads. Les fonctions *EnterCriticalSection()* et *LeaveCriticalSection()* imposent aux threads d'accéder séquentiellement aux ressources partagées.

- Le Mutex autorise un accès exclusif à une ressource par un thread.
- Le Sémaphore contrôle le nombre de threads utilisant simultanément une même ressource.
- L'Evénement (Event) permet d'avertir un ou plusieurs threads qu'un événement attendu s'est produit de façon qu'ils effectuent une action donnée.

Un Mutex, un Sémaphore ou un Evénement sont des objets gérés par le système d'exploitation. Ils reçoivent un nom leur servant d'identification et peuvent être dans un des deux états suivants : signalé ou non signalé. On appelle la fonction *WaitForSingleObject()* au

moyen d'un thread pour attendre qu'un objet de synchronisation soit en état signalé. Si c'est le cas, l'attente se termine et le thread peut accéder à la ressource protégée par l'objet. S'il est en état non signalé, le thread est bloqué jusqu'à ce qu'un autre thread remette l'objet en état signalé ou jusqu'à ce qu'un temps d'attente soit dépassé. Le fait d'être bloqué sur l'appel d'une fonction n'empêche pas le système de passer la main aux autres threads.

III. 3 Mémoire Partagée

L'autre problème qui se pose pour la réalisation d'une application multitâche est l'échange ou le partage de données. Windows NT se caractérise par une grande protection de la mémoire, ce qui fait que chaque processus dispose de son propre espace d'adressage. Les adresses en mémoire sont alors dites virtuelles, ce qui signifie que l'adresse 1792 dans un processus A, ne correspond pas au même emplacement physique que l'adresse 1792 du processus B. De cette manière, un processus n'a aucun moyen d'accéder aux données d'un autre processus. Si cela représente un avantage certain pour la sécurité et la fiabilité du système, il peut néanmoins constituer un handicap lorsqu'il est nécessaire de partager des données.

Le partage de données entre deux processus différents se fait, dans Windows NT, à l'aide des fichiers mappés (mapped files). Un fichier mappé est un espace d'adressage virtuel auquel plusieurs processus peuvent accéder simultanément. La fonction *CreateFileMapping()* crée un fichier mappé dont la taille peut être importante. Lors de l'appel de cette fonction, il n'y a pas d'allocation de mémoire physique, mais simplement la mise en réserve d'un ensemble d'adresses virtuelles. Ce n'est que lors de l'appel de la fonction *ViewMapOfFile()* qu'une partie de la mémoire réservée est allouée et que le thread peut y accéder. Au moment de sa création avec la fonction *CreateFileMapping()*, le fichier mappé reçoit un nom ; comme les autres threads connaissent ce nom, ils peuvent y accéder.

Une autre utilisation des fichiers mappés permet d'accéder à un fichier comme s'il s'agissait d'une zone située en mémoire.

Chapitre 4

Description du Prototype de communication SDDS RP*

I. L'Architecture Client/Serveur

Le prototype de communication SDDS RP* est conçu, comme la plupart des applications réseau, en supposant qu'un côté se comporte comme un client et l'autre comme un serveur. La partie serveur de l'application a pour fonction de procurer des services définis pour les clients.

Les serveurs sont catalogués en deux classes : itératifs et concurrents :

- Un serveur itératif déroule la séquence suivante :
  - I1. Attendre l'arrivée d'une requête émise par un client.
  - I2. Exécuter la requête du client.
  - I3. Retourner la réponse au client qui a émis la requête.
  - I4. Retourner à la phase I1.

  Ce mode de fonctionnement itératif pose un problème lorsque la phase I2 prend du temps, car durant ce laps de temps aucun autre client ne peut être servi.

- Un serveur concurrent adopte le fonctionnement suivant :
  - C1. Attendre l'arrivée d'une requête émise par un client.
  - C2. Démarrer un nouveau serveur pour traiter la requête du client. Ceci peut impliquer la création d'un nouveau processus, d'une nouvelle tâche ; ce qui dépend du système d'exploitation utilisé. Ce nouveau serveur prend en charge l'intégralité du traitement de cette requête
  - C3. Retourner à la phase C1.

Notre prototype de communication adopte un fonctionnement concurrent en se base sur la mise en œuvre du multitâche sous Windows NT grâce à l'utilisation des threads.

II. L'application SDDS-RP* / Client

Le rôle de l'application client est de permettre à des applications de formuler des requêtes sur le fichier SDDS comme si celui-ci se trouvait sur disque, rendant ainsi transparente l'émission des requêtes vers les sites serveurs et la réception des réponses. Les applications destinées à manipuler le fichier SDDS vont donc faire appel à des routines implémentées au niveau des applications client SDDS-RP.

Pour cette première version, les requêtes sont générées directement par l'application client. Pour les versions ultérieures, il faudra prévoir une interface entre les applications qui accèdent au fichier SDDS et le module client du protocole de communication SDDS-RP*. L'interface permettra d'appeler le protocole SDDS-RP* de la même façon qu'une ressource système.

L'application Client est composée de deux modules de traitement qui tournent en parallèle : le premier écoute les messages en provenance des applications et le second, les messages en provenance des sites serveurs.

Le module d'écoute des messages émises par les applications adopte le fonctionnement suivant :

E1 - Attendre l'arrivée d'une requête émise par une application ;

E2 - Chercher l'adresse du serveur adéquat ;

E3 - Envoyer la requête vers ce serveur ;

E4 – Retourner à l'étape E1.

Le module d'écoute des réponses émises par les applications serveurs SDDS-RP* adopte le fonctionnement suivant :

A1 - Attendre l'arrivée d'une réponse à une requête ;

A2 – Mettre à jour l'image du client sur le fichier ;

A3 – Vérifier à quelle application appartient la réponse reçue ;

A4 – Retourner la réponse à l'application trouvée ;

A5 – Retourner à l'étape A1.

Les principales fonctions de SDDS-RP* / Client

Recevoir_Requete() : Cette fonction permet de lire une requête passée par une application. Elle attribue un identificateur unique à la requête (Id_Req) et maintient une table (Id_Req, Id_App). Id_App est l'identificateur de l'application appelante. Avec Id_Req, l'application

sera capable de reconnaître la réponse correspondant à chaque requête. La table (Id_Req, Id_App) permettra de remettre à chaque application la réponse correspondant à sa requête. C'est particulièrement important dans le cas où une application enverrait plusieurs requêtes en même temps.

ChercheAdresseServeur(algo, fichier, clé) : Cette fonction permet de calculer l'adresse du serveur auquel doit être envoyé la requête passée par l'application. Ce calcul dépend de la SDDS spécifiée par l'application. En entrée, on doit spécifier l'algorithme (RP*N, RP*C, etc.), le fichier et la clé.
Pour RP*N, cette fonction retourne l'adresse multicast associée au fichier spécifié en paramètre. Pour RP*C, le calcul consiste à parcourir l'image du client sur le fichier et à rechercher le serveur dont la portée de la case recouvre la clé contenue dans la requête. Le résultat fourni est l'adresse IP de la machine sur laquelle tourne ce serveur.

EnvoyerRequete(Id_requete, requête, Id_Serveur) : Cette fonction se charge d'envoyer la requête en utilisant l'adresse socket retournée par la fonction *ChercheAdresseServeur().*

Reception_reponse(Id_requete) : Cette fonction consiste en un thread qui reste à l'écoute des réponses envoyées par les sites serveurs. Une réponse peut être reçue d'un serveur différent de celui auquel on a envoyé la requête dans le cas de redirection. La réponse d'un serveur doit donc être munie de l'identificateur de requête Id_requete. Si les données de la réponse sont volumineuses, le serveur retourne alors une adresse socket TCP à laquelle le client devra se connecter pour récupérer la réponse.

Retourner_reponse(Id_App, Id_requete, Réponse) : Cette fonction doit retourner la réponse à l'application initiatrice de la requête.
MajImage_Client(CleMin, CleMax, Adresse) : Si le client utilise l'algorithme RP*C, alors cette fonction se chargera d'extraire le champ IAM de la réponse du serveur et mettra à jour l'image du client sur le fichier SDDS.
Initialise_Client() : Cette fonction se chargera de démarrer le service client SDDS-RP*. Il s'agit de :

- Créer les sockets de communication ;
- Créer le fichier mappé qui constituera la file d'attente des requêtes en attente de réponse ;
- Lancer le démon d'écoute.

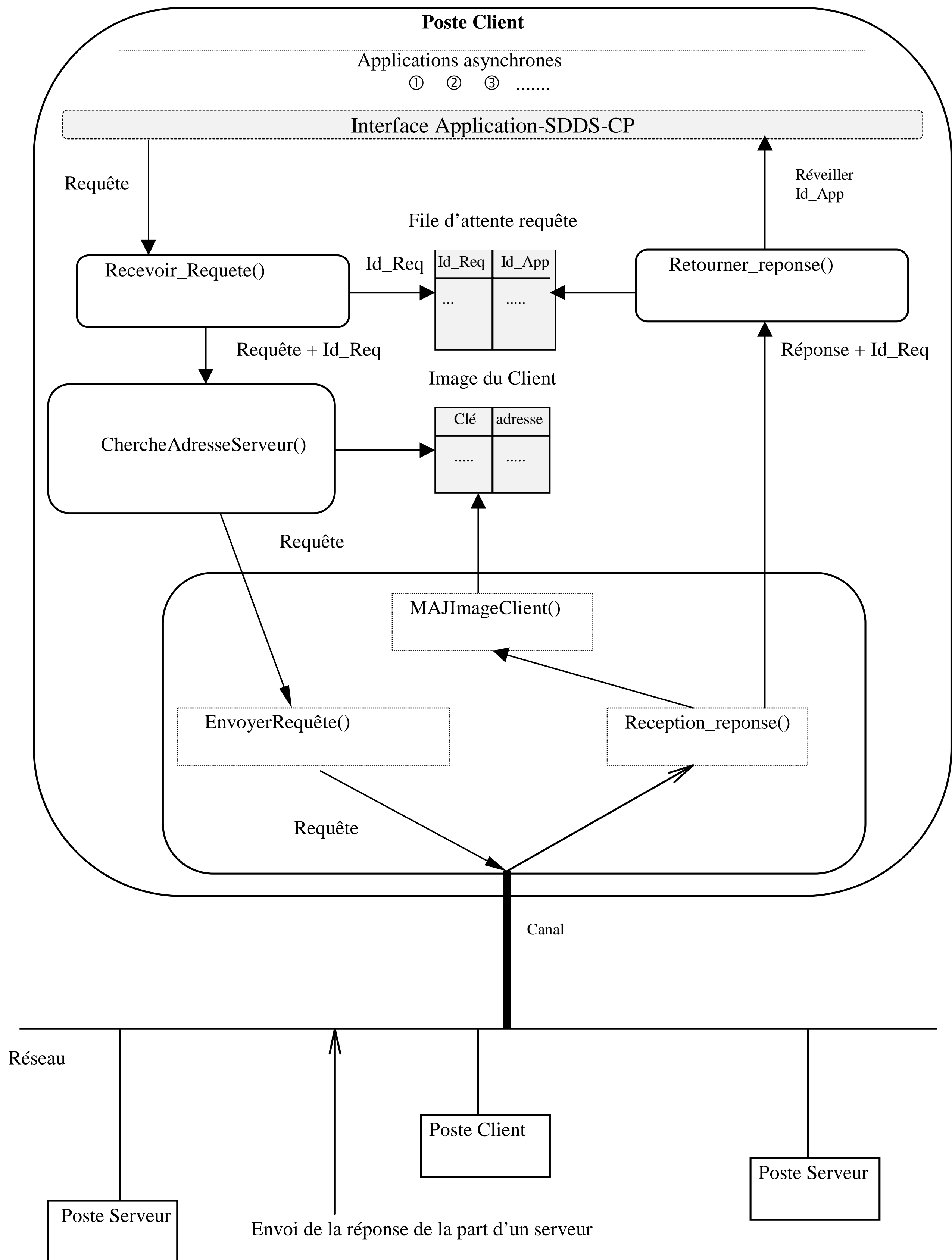


Figure 1 : SDDS-RP : Client*

III. L'application SDDS-RP* / Serveur

Le rôle des applications serveurs est de recevoir les requêtes émises par les applications Client SDDS-RP*, d'effectuer le traitement demandé sur le fichier SDDS et de retourner la réponse. C'est l'application serveur qui accède au fichier SDDS stocké en mémoire vive.

Le serveur SDDS-RP* est composé d'une partie Communication qui se charge de recevoir les requêtes émises par les sites client et de retourner les réponses une fois le traitement effectué. Une autre partie dite Traitement SDDS se chargera d'analyser et d'effectuer les traitements demandés dans la requête.

Conçu pour servir un grand nombre de clients, le programme serveur utilise un thread qui se charge de mettre les requêtes clients reçues dans une file d'attente. Le serveur maintient également une file d'attente de thread-de-travail dont le nombre est fixé en fonction de la charge au niveau du serveur.

Le thread d'écoute adopte le fonctionnement suivant :

E1 - Attendre l'arrivée d'une requête émise par un client SDDS-RP* ;

E2 - Mettre la requête dans la file d'attente et signaler l'événement ;

E3 - Retourner à l'étape E1.

Un thread de travail adopte le fonctionnement suivant :

T1 - Attendre que l'événement arrivée requête soit signalé ;

T2 - Prendre une requête de la file d'attente ;

T3 - Analyser la requête pour voir le traitement demandé ;

T4 - Selon le cas, lancer le traitement local, la redirection de la requête ou l'éclatement de la case ;

T5 - Retourner à l'étape T1.

Les principales fonctions de SDDS-RP* / Serveur

ThreadEcoute() : Cette fonction correspond au thread d'écoute. Elle se charge de :

- Ecouter au niveau de la socket de communication ;
- Placer les requêtes reçues des clients dans la file d'attente ;
- Signaler l'événement arrivée requête.

ThreadTravail() : Cette routine correspond à la tâche faite par un Thread-de-travail :

- prendre une requête de la file d'attente
- appeler *Analyse_Requete()*
  - Si traitement local alors : appeler le processus *Trait_Local()*.
  - Si redirection alors : appeler le processus *Trait_Redirection()*.
  - Si éclatement alors : appeler le processus *Trait_Eclatement()*.

Trait_Local() : Ce processus doit :

- Faire le traitement adéquat (insertion, recherche, suppression, modification). C'est à ce niveau que se fera la jonction avec l'organisation interne d'une case SDDS ;
- Mettre la réponse dans un buffer ;
- Appeler la fonction *EnvoyerReponse()*.

Trait_*Redirection()* : Cette fonction est appelée si une requête est reçue par message unicast avec une clé n'appartenant pas à la portée de la case. Il s'agit alors d'ajouter l'adresse et la portée du serveur au message initial et d'envoyer le tout en multicast aux autres serveurs.

Trait_Eclatement() : Ce processus est déclenché quand la case a atteint le nombre maximun d'enregistrements qu'elle peut contenir. Il s'agit de :

- Envoyer un message de recherche de serveur disponible ;
- Dès réception d'une réponse, se connecter au serveur disponible ;
- Déplacer la moitié des enregistrements de la case en débordement vers la nouvelle case ;
- Mettre à jour la portée de la case initiale.

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

Envoyer_Reponse() : Si les données ne sont pas volumineuses, la réponse est envoyée au client par message UDP. Sinon, le serveur crée une socket TCP et envoie l'adresse au client par message UDP. A la réception de ce paquet, le client se connecte à l'adresse TCP pour récupérer les données de la réponse.

<u>Initialise_Serveur() :</u> Cette fonction se chargera de démarrer le service serveur SDDS-RP*. Il s'agit de :

- Créer les sockets de communication ;
- Créer le fichier mappé qui constituera la file d’attente des requêtes en attente de traitement ;
- Lancer le thread d’écoute ;
- Lancer les threads de travail.

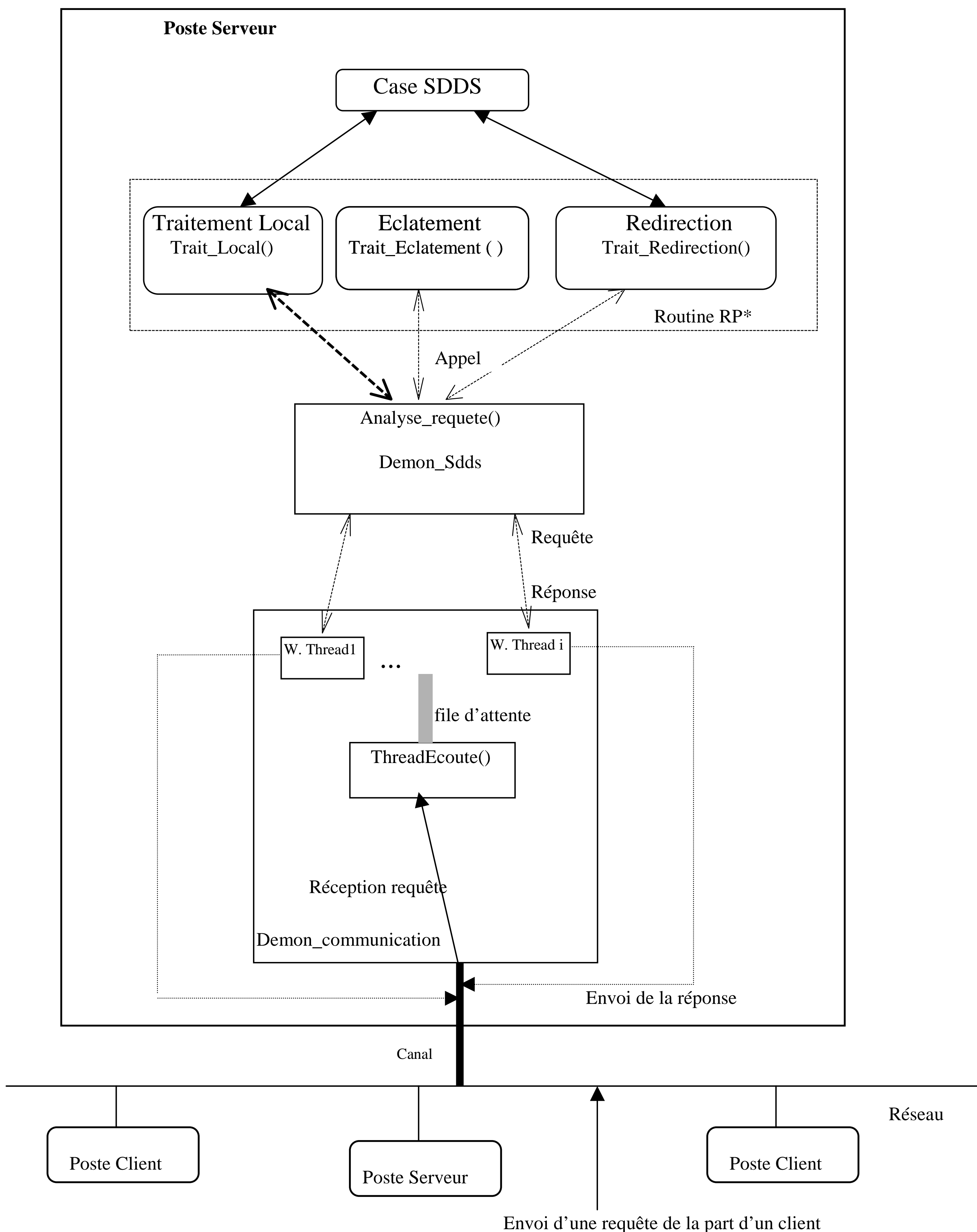


Figure 2 : SDDS-RP : Serveur*

IV. Mesure de performances

Le temps de traitement d'une requête sur un fichier SDDS est fonction de l'algorithme utilisé et des performances du réseau et du CPU. Les performances d'accès d'une SDDS sont mesurées en nombre de messages par opération afin que les résultats soient indépendants du débit du réseau et de la vitesse du CPU.

La particularité des algorithmes RP*N est qu'ils ne nécessitent pas d'image du fichier sur les sites clients ; donc, il n'existe pas d'erreur d'adressage, ni de messages correctifs IAM. Les opérations de modification, d'insertion et de suppression d'un enregistrement nécessitent un seul message multicast. La recherche d'une clé nécessite deux messages : un multicast et un point-à-point pour la réponse. Dans l'algorithme RP*N, les opérations sur un enregistrement sont indépendantes du nombre de cases qui constituent le fichier.

Les performances d'accès d'une SDDS RP*C sont liées à l'état de l'image du client sur le fichier. Un nouveau client produira des erreurs d'adressage jusqu'à la convergence de son image. Les opérations de recherche, d'insertion, de modification et de suppression nécessitent alors trois messages (requête, redirection multicast, réponse). La rapidité de convergence dépend du nombre de cases qui constituent le fichier.

Pour un client avec une image correcte, les opérations de recherche, d'insertion, de modification et de suppression nécessitent deux messages point-à-point (requête, réponse).

Tests de validité

Les tests portent sur un fichier de 1000 enregistrements de 100 octets. Il s'agit de mesurer la durée de la lecture séquentielle des 1000 enregistrements du fichier s'il est stocké sur disque et s'il est mappé en mémoire RAM locale. Les tests réseau consistent à mesurer la durée de 1000 messages aller-retour entre des machines distantes en unicast et en multicast.

Les résultats que nous avons obtenus sur notre réseau 10Mbits montrent une sensibilité par rapport à la machine(processeur) et au disque (plus ou moins rapide). En moyenne, on obtient pour 1000 lectures 3094 ms sur disque local, 1082 ms sur fichier mappé en memoire locale. Pour les messages entre machines distantes, on obtient pour 1000 messages aller-retour: 1672 ms en multicast et 1622 ms en unicast.

On constate systématiquement que le temps d'accès sur disque local est supérieur à la somme du temps d'accès sur fichier mappé et du temps aller-retour des messages entre clients et serveurs distants.). Cependant, nous espérons de bien meilleurs résultats sur un réseau 100Mbits pour corroborer notre thèse.

Conclusion

Notre objectif été de concevoir un protocole de communication et de programmer un prototype de système de communication entre les serveurs et les clients des SDDS RP*N et RP*C en choisissant le modèle des sockets Windows comme interface de communication entre les machines distantes et la famille TCP/IP comme protocole réseau.

Pour mieux exploiter la puissance des machines, nous avons utilisé les techniques de programmation avancées sous Windiws NT notamment la programmation réseau avec les Windows sockets, la programmation parallèle avec l'utilisation des threads, la synchronisation de processus, la gestion de la mémoire avec la technique des fichiers mappés.

Le travail de documentation sur les SDDS et les systèmes client/serveur nous a permis d'approfondir nos connaissances sur les systèmes répartis et de nous familiariser avec le système d'exploitation Windows NT.

Ce travail entre dans le cadre du projet global de l'équipe SDDS de Dakar qui consiste à l'implémentation et la mesure de performances de toute la famille des SDDS RP* : RP*N, RP*C et RP*S. Le prototype ainsi programmé est en cours d'intégration avec l'organisation interne d'une case SDDS RP* déjà développée et les temps de réponse obtenus vont permettre de valider les calculs théoriques de performances d'accès d'une SDDS.

Enfin, notre travail ouvre l'horizon à plusieurs directions de recherche. De nombreux problèmes méritent une étude spécifique tels que la consistance des données, la sécurité d'accès aux fichiers SDDS, les opérations complexes sur des fichiers SDDS (jointure, fonctions agrégat...).

Références

[DIENE98] : Aly W. Diene. Organisation interne d'une case SDDS RP*, Mémoire de DEA, Dept. Mathématiques ET Informatique, Université Cheikh Anta Diop de Dakar, fevrier 1998.

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

[LITWI80] : Litwin, W. Linear Hashing : a new tool for file and tables addressing. Reprint from VLDB-80.

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

[LITWI93a] : 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.

[LITWI95b] : Litwin, W. et Neimat, M. 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.

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

[NOW96] : "http://http.cs.berkeley.edu/projects/" Network of Workstations. Building system support for using a network of workstations as a distributed supercomputer on a building-wide scale.

[SAHLI96] : Système de Communication (SDDS-CP) entre les serveurs et les clients d'une SDDS LH*, *Lab. GERM, Université Paris IX Dauphine*, 1996.

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

[SOULE95] : R. SOULEÏMAN*, Rapport de stage du DEA 127, Lab. GERM, Université Paris IX Dauphine,* September 1995.