Laboratoire ID

Composition de service de données et de méta-données dans un système de fichiers distribué

Adrien Lebre

sous la direction de
Philippe Augerat

Membres du jury :
Joëlle Coutaz
Jean Marc Vincent
Toan Nguyen
Renaud Lachaize

Juin 2002

INRIA
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

GRENOBLE 1
UNIVERSITÉ JOSEPH FOURIER
SCIENCES,TECHNOLOGIE,MÉDÉCINE

Composition de service de données et de méta-données dans un système de fichiers distribué

Adrien Lebre
sous la direction de Philippe Augerat

{adrien.lebre,philippe.augerat}@imag.fr
Laboratoire ID-IMAG
Montbonnot St Martin, Isère

Résumé

La gestion des fichiers est, depuis fort longtemps, une composante clé au sein d'un système informatique. Dépendantes de l'architecture sous-jacente, les contraintes imposées aux différents modèles de gestion de fichiers n'ont cessé d'évoluer (méthodes d'accès locales, partage de ressources dans un LAN, au sein d'un cluster ou encore échange d'informations sur le *web*). Les systèmes de fichiers doivent s'adapter afin de proposer des solutions permettant d'exploiter au maximum les capacités proposées par les plate-formes actuelles (ressources importantes de stockage, agrégation des débits, puissance d'analyse) et ceci en tenant compte des diverses contraintes (passage à l'échelle, hétérogénéité, tolérance aux fautes ...).

Ce rapport présente, dans un premier temps, différentes solutions répondant seulement à certains des problèmes évoqués ci-dessus. Ces diverses techniques, souvent complexes à mettre en œuvre, sont construites sur des architectures matérielles dédiées. Une solution plus conviviale, fondée sur le standard NFS est développée depuis un peu plus d'un an au sein du laboratoire ID. C'est sur ce nouvel outil, décrit dans une seconde partie, que nous avons pu étudier les limites et les performances de la distribution et de la duplication des serveurs au sein d'une grappe. Les résultats obtenus s'avèrent être très prometteurs pour réussir un passage à l'échelle caractérisé par une exploitation optimisée des ressources disques ainsi qu'une utilisation intelligente des performances fournies par les réseaux sous-jacents.

Mots clés : Cluster, système de fichiers distribué, NFS, *spoofing*.

Table des matières

Table des figures

Remerciements

Je tiens à remercier tout d'abord Philippe Augerat qui m'a donné la possibilité de mener cette étude au sein du laboratoire Informatique et Distribution, de son encadrement précieux et de l'ensemble du temps dont il a eu l'amabilité de me consacrer.

Je remercie également l'ensemble des chercheurs (thésards et permanents) pour les conseils et les explications qu'ils ont su me fournir. Plus particulièrement, Pierre Lombard, qui m'a maintes fois aidé dans la compréhension du système NFSp et qui a contribué énormément à la rédaction de ce rapport.

Les rapporteurs et les membres du jury qui me font l'honneur de juger ce travail.

Enfin, je remercie l'ensemble du personnels présent dans le laboratoire pour leur accueil chaleureux.

Introduction

Un véritable engouement est apparu autour des techniques visant à rendre transparente et performante l'utilisation, pour le calcul, de centaines d'ordinateurs personnels (PC) inter-connectés par un réseau local. La principale raison est que ce type d'architecture, communément appelé grappe[1] ou encore *cluster*[SSB$^+$95], permet de fournir une puissance équivalente à celle des grands super-calculateurs et ceci pour un coût nettement inférieur.

De nombreux travaux scientifiques ont été et sont actuellement menés dans ce domaine. Ils tentent de répondre aux problèmes et contraintes qui sont liés à ce nouveau type d'environnement d'exécution concernant l'ordonnancement, l'algorithmique parallèle, la programmation ainsi que l'accès et le partage à distance de l'ensemble des ressources.

Conséquence des performances atteintes dans ce champ, le nombre de machines[2] composant le *cluster* ne cesse de croître. Ainsi, l'emploi d'un protocole fondé sur un serveur de stockage unique (à la NFS, protocole maintes fois éprouvé au sein des LANs) est devenu pratiquement impossible dans ce genre de contexte (passage à l'échelle, débit) et ne permet pas d'exploiter la totalité du potentiel disponible (espace disque[3], équilibrage de charge). A l'opposé de techniques matérielles de stockage où un réseau rapide (giga-ethernet, fibre optique...) permet à plusieurs noeuds (mais toujours en quantité limitée) de partager efficacement un espace de stockage centralisé, des solutions fonctionnant au dessus des protocoles de l'Internet ont été proposées pour lesquelles un nombre éventuellement important d'entités fournit un accès contrôlé à ses fichiers. Napster ou encore les systèmes pair à pair comme Gnutella ou Freenet sont les exemples les plus connus d'implémentation de partages de fichiers (principalement multimédia) à l'échelle du web.

Par ailleurs, l'extension des grappes aux grilles [FK90] (interconnexion de plusieurs architectures de calculs par un réseau dédié comme VTHD) a fait surgir de nouveaux paramètres tels que la sécurité et l'hétérogénéité des différentes «ossatures». En effet, une application distribuée sur une grille doit accéder conjointement à plusieurs Intranets en tenant compte des multiples propriétés des réseaux, des configurations et des principales fonctionnalités de chaque sous système. De plus, le fait que certaines structures ne soient accessibles qu'à des périodes spécifiques (utilisation de réseau d'entreprise la nuit ou pendant les congés) entraîne là aussi, d'importantes com-

[1]Le laboratoire ID possède, depuis 2001, une grappe de 225 PC iVectra 750 MHz, 256Mo RAM et 15Go de stockage IDE.

[2]Par la suite, nous utiliserons le terme de noeud pour exprimer une élément de la grappe.

[3]Un noeud du cluster ID nécessite un système de base d'environ 4Go ce qui laisse prés de 11Go inutilisés, soit plus de 2To pour l'ensemble de l'architecture

plications (découverte dynamique de ressources, résilience ...) en plus d'un nouveau changement d'échelle en terme de nombre d'ordinateurs.

Les projets comme Seti@home (vol de cycles pour le calcul de FFT, 2.3 millions de personnes inscrites, machine «virtuelle» la plus puissante au monde) ou encore Xtrem Web en France (plate-forme tentant d'abstraire les grilles afin de fournir un support de calcul ultra-performant aux industriels) ont dévoilé de nombreux problèmes liés entre autre au partage de ressources à très grande échelle (représentation, nommage et cohérence des informations). Les disparités des systèmes d'exploitation (Windows, UNIX, Macintosh) présentent dans de telles architectures réparties, complexifient d'autant plus la conception. Toutefois, il est presque certain que le développement des applications futures va s'appuyer sur ces grandes disponibilités de puissance de calcul et d'espace de stockage quasi-infini ; la mise en œuvre d'outils performants est donc impérative afin de proposer des supports fiables et puissants pour l'avenir permettant de faire intéragir calculs et échanges d'informations.

Nous allons effectuer, après avoir présenté la terminologie spécifique aux systèmes de fichiers, une description succincte de la majeure partie des systèmes de fichiers distribués actuels. Par la suite, nous nous intéresserons, à une étude plus exhaustive du modèle NFSp [eYD02] [LD02] développé au laboratoire ID. L'extension de ce système à plusieurs serveurs distribués et/ou dupliqués (travail proposé dans le cadre du DEA), y sera largement développée. Les performances et les limites obtenues apparaîtront dans une dernière partie.

Enfin, nous énoncerons, durant la conclusion, les diverses perspectives entrouvertes par l'étude menée et le vif intérêt d'une gestion dynamique (à la volée) des mouvements des données en fonction des possibilités du réseau et de la disponibilité des serveurs.

Première partie

Les systèmes de fichiers distribués

Chapitre 1

Terminologie

L'informatique a fourni le moyen de traiter rapidement et de conserver des informations dans de nombreux domaines d'activités : industriel, économique ou social. La quantité de données générée par certaines applications modernes (climatologie, biologie moléculaire, génomique, physique des hautes énergies, traitements d'images ...) s'avère gigantesque. Une gestion performante du stockage, à court et à long terme, se révèle être un paramètre prépondérant.

Comme dans les systèmes locaux, le travail d'un gestionnaire de fichiers en environnement distribué est de classer les programmes et les données pour les rendre accessibles à la demande et ce en tout point de l'architecture. L'ensemble de la littérature abordant ce sujet s'appuie sur une terminologie spécifique afin de caractériser les diverses notions et techniques utilisées. Nous allons la présenter tout au long de ce chapitre.

1.1 Généralités

Partie intégrante du système d'exploitation, un système de fichiers distribué peut le plus souvent être fonctionnellement divisé en deux parties distinctes : le système de fichiers proprement dit et le service de gestion. Le premier prend en charge les opérations sur les fichiers, comme la lecture, l'écriture ou l'ajout, alors que le second crée et gère les répertoires (ajout, destruction de fichiers, parfois appelé service de nommage). Cette notion a d'ailleurs été largement utilisée pour développer NFSp, système de fichiers développé au laboratoire ID (cf. chapitre 3).

Afin de bien visualiser toutes les entités qui composent un tel système, nous préciserons en ajoutant qu'un système de fichier fournit un ensemble de services à des clients (les fonctionnalités étant composées, en majeure partie, des primitives de création, de destruction, de lecture et d'écriture).

Les systèmes de fichiers distribués sont évalués par plusieurs facteurs : réactivité, transparence, disponibilité ou encore possibilité du passage à l'échelle[1]. Le terme de réactivité évoque la quantité de temps nécessaire pour satisfaire une requête (latence réseau, analyse de la demande et formulation de la réponse). Les autres caractéristiques sont développées ci-après.

[1]Par la suite, nous utiliserons l'anglicisme *scalabilité* pour définir cette propriété.

1.2 Transparence

Une des premières propriétés fondamentales d'un DFS[2] est sa transparence vis à vis du réseau. Les clients doivent être capable d'accéder à des ressources distantes en utilisant les mêmes primitives d'opérations sur les fichiers que celles applicables aux fichiers locaux.

Nous rappellerons ici une notion importante dans l'implémentation d'un système de fichier [Bac86] :

Le concept de nommage est la relation entre le nom logique et l'identifiant physique.

Un usager se réfère à un fichier par un nom textuel. Une correspondance est alors réalisée avec un identifiant numérique (appelé *inode* ou bloc de méta-données) qui est lui même une interprétation d'une portion (bloc) d'un disque physique. Ces abstractions permettent de cacher à l'utilisateur final les détails de l'implantation physique du fichier sur le périphérique de stockage. Pour un DFS, il est impératif d'ajouter une nouvelle dimension afin de rendre transparente la localisation du fichier au sein du réseau.

Ainsi deux niveaux de transparence ont été définis pour caractériser un DFS :

Transparence de situation, le nom du fichier ne comporte aucune information sur son emplacement physique.

Indépendance à la mobilité, le nom du fichier n'a pas à être modifié lorsque sa situation géographique au sein du réseau évolue.

Le niveau d'abstraction qu'offre l'indépendance à la mobilité est le plus élevé. Il permet, entre autre, de réaliser de l'équilibrage de charge ou de répliquer des données entre les différentes unités de stockage. Le service de gestion peut dynamiquement faire migrer des fichiers d'un point du système vers un autre, ceci dans le but de réduire le taux de charge d'un serveur. La réplication des fichiers (à un nom correspond plusieurs « images » physiques) apporte un plus grande disponibilité et une meilleur tolérance aux pannes (cf. paragraphe 1.5).

Trois approches de nommage sont principalement utilisées dans les DFS afin d'apporter la transparence.
La plus simple permet uniquement de garder les mêmes appels de procédures pour les accès aux fichiers locaux ou distants. Il s'agit d'ajouter au nom du fichier, la machine sur laquelle il est stocké. Le nom complet du fichier correspond alors au tuple «`hôte : nom`».
Une seconde solution consiste à importer une certaine partie d'un espace distant au sein de son propre espace de nom. Ce type d'opération est traditionnellement connu grâce à la commande `mount` d'UNIX. Elle fournit une transparence de situation mais comporte certains désavantages. Par exemple, il est tout à fait possible d'obtenir deux structures de fichiers différentes entre plusieurs entités de l'architecture.
L'utilisation d'un unique espace de nom global pour l'ensemble du système permet d'apporter le plus haut niveau de transparence. Malheureusement, l'implémentation technique d'un tel espace est complexe. En effet, certains fichiers sont spécifiques à une entité du système (pour les nœuds utilisant un noyau UNIX par exemple, le répertoire `/dev` contient tous les fichiers correspondant aux périphériques d'entrées/sorties).

[2]*Distributed File System*

1.3 Sémantique de partage des fichiers

Ce critère est important dans l'évaluation d'un système de fichier autorisant plusieurs accès concurrents à des fichiers partagés. Cette propriété définit comment et quand les modifications, effectuées par un nœud, sont observables par l'ensemble des machines distantes. Il est impératif de connaître réellement ce comportement. En effet de nombreux problèmes d'intégrité des données peuvent survenir lors d'accès simultanés sur une même ressource par plusieurs clients.

Quatre sémantiques ont été définies afin de classer les diverses possibilités :

Sémantique fondée sur le modèle UNIX, toute modification qui survient sur un *bloc d'un fichier* [3] est vue instantanément sur l'ensemble des nœuds accédant à l'information au même moment. En d'autres termes, il n'existe qu'une seule image unique du fichier à un instant donné dans tous l'environnement.

Sémantique de session, une session est définie comme l'ensemble des opérations exécutées entre l'ouverture et la fermeture d'un fichier par un client. Ainsi, cette sémantique stipule que lorsqu'une session est ouverte sur un nœud, toute machine distante sera avisée des changements uniquement lors de la fermeture de celle-ci sur le précédent nœud. Par ailleurs, il est précisé qu'au sein d'une même entité, la sémantique UNIX sera appliquée. Un fichier peut donc temporairement avoir plusieurs images réparties sur l'architecture. Le désavantage est alors qu'un client puisse réaliser une lecture erronée du fichier.

Sémantique dite « immuable », lorsqu'une ressource devient partagée (des clients souhaitent accéder à la même information), le fichier est alors utilisable uniquement en lecture. La mise en place d'un verrou rend toute modification impossible.

Sémantique « transactionnelle », toute session sur un fichier est une transaction (au sens base de données). Dès qu'un client ouvre un fichier, celui ci est verrouillé et aucun autre nœud ne peut y accéder jusqu'à sa fermeture.

Les deux premières sémantiques sont celles qui sont le plus représentées dans les modèles de partage de fichiers. Elles obligent les développeurs à tenir compte des problèmes d'exclusions mutuelles sur les ressources. Cependant, elles proposent un réel parallélisme des accès sur l'ensemble des données et donc un gain considérable au niveau des performances. Les deux dernières intègrent des solutions de synchronisation mais peuvent entraîner des phénomènes de famine et d'interblocâge. Nous verrons par la suite (cf. paragraphe 2.3.2) que des systèmes de verrous, plus élaborés, peuvent être utilisés à un grain plus fin. Un verrou est placé sur une portion du fichier et non sur sa globalité. Cette manière fournit un moyen de résoudre les difficultés liées aux accès concurrents tout en laissant la possibilité de paralléliser les opérations.

1.4 Méthodes d'accès à distance et gestion de cache

Selon la sémantique appliquée, il peut s'avérer performant de réaliser une gestion de cache sur chaque nœud de l'architecture. Le fait d'accéder à une image locale des données permet d'éviter, d'une part, le transfert de messages sur le réseau et d'autre part une saturation potentiellement inutile du nœud stockant la ressource.

[3] Le terme de bloc équivaut à la granularité du système de fichier. C'est la plus petite portion que le DFS peut remettre lors d'une requête.

Néanmoins, la mise en œuvre d'un module de cache, afin d'obtenir de réels gains, requiert l'analyse de différents points :

Emplacement, le cache peut soit être stocké en mémoire vive soit sur le disque local. Si un cache, en mémoire vive, propose un accès beaucoup plus rapide à l'information, il sera perdu lors d'une éventuelle défaillance du nœud et devra par conséquent être reconstruit lors du retour à la normale.

Granularité, le surcoût total du trafic de données sur le réseau varie selon la granularité du cache. Si pour une unique requête, le serveur retourne une portion du fichier plus grande, la probabilité que le prochain accès, par le même processus client, soit dirigé vers des données incluses dans ce bloc augmente. Ce qui se révèle être le plus souvent le cas lors de la lecture d'un fichier. De plus, il est plus rapide pour le serveur de retourner une quantité d'informations contiguë que plusieurs paquets correspondant à différentes positions au sein du disque. Cependant, plus ce facteur est grand, plus les opérations de mise à jour peuvent se révéler coûteuse selon la sémantique employée.

Mise à jour et cohérence, c'est le véritable problème de la gestion d'un cache de données. La validation d'un cache peut soit être à l'initiative du demandeur soit du serveur. Dans le premier cas, une requête de validation est transmise au serveur soit périodiquement soit à l'ouverture ou à la fermeture d'une ressource. La seconde approche oblige le serveur à connaître l'ensemble des informations cachées par chaque machine (cf. paragraphe 1.5). Il transmettra des messages d'invalidation ou de mise à jour à chaque nœud concerné et ceci pour chaque modification.
Comme nous l'avons cité auparavant, le coût généré par cette mise à jour dépend totalement de la sémantique implémentée. La sémantique UNIX entraînera un trafic important (chaque modification devra invalider les caches, voire les mettre à jour). Une sémantique de session sera plus qu'avantageuse lors de l'utilisation de fichiers temporaires (des informations qui n'ont pas besoin d'être physiquement sur un serveur distant et qui engendreraient un trafic inutile sur le réseau).

Ainsi, il est nécessaire de trouver un bon compromis entre la granularité, la sémantique utilisée et les ressources matérielles disponibles afin d'obtenir un réel gain. La plupart des modèles proposent différentes techniques selon le service demandé (création, lecture, écriture), le nombre d'accès concurrents, le type de ressources (répertoire, fichiers) et les paramètres cités ci-dessus.

1.5 Tolérance aux pannes

La tolérance aux pannes est un important et large problème dans les systèmes informatiques (toute une littérature y est d'ailleurs consacrée). Perte de réseau, dysfonctionnement du programme, arrêt d'une des machines, altération d'une unité de stockage, tous doit être pris en considération afin de permettre au DFS de continuer à s'exécuter dans un état cohérent. Un système qui requiert un arrêt lors d'une quelconque erreur n'est pas tolérant aux fautes. De manière simplifiée, toute les solutions, permettant d'améliorer ce facteur, gravitent autour de la redondance matérielle et/ou logicielle.

Nous allons, tout d'abord, nous attarder sur un point rapidement énoncé dans le paragraphe précédent. En effet, la gestion du cache qui permet d'augmenter les performances en terme de réactivité, complexifie considérablement l'implémentation d'un service robuste.

Lorsque les nœuds «fournisseurs» stockent, en mémoire vive, des renseignements sur chaque connexion (descripteur de fichier en cours ...) avec les noeuds «demandeurs», le terme de *serveur avec état* est défini. Cette technique permet d'optimiser le transfert d'informations, mais comporte les défauts de ses qualités : lors de la perte des informations concernant les clients, un retour à un état normal peut s'avérer très pénalisant (un dialogue avec chaque noeud est engagé pour retrouver la totalité des informations stockées avant la panne). De plus, le serveur doit être averti en cas d'arrêt de toute entité de l'architecture afin de désallouer l'espace mémoire contenant les données devenues invalides.

A l'opposé, la terminologie de *serveur sans état* est employée pour un nœud totalement indépendant des autres machines. Un serveur sans état permet de supprimer nombreux problèmes de tolérance aux pannes. Tout d'abord, un serveur pourra sans aucune difficulté réintégrer son état après une éventuelle erreur. Un mauvais fonctionnement d'un client n'engendrera aucune perte de performances dans le système. D'autre part, un client ne se préoccupe pas de l'état du serveur puisqu'il réémmettra sa requête jusqu'à satisfaction. Cependant un tel service impose l'utilisation de messages plus longs et offre une réactivité bien moindre que celle fournie par un serveur avec état.

Parmi les autres critères de la tolérance aux fautes, deux concepts caractérisent l'état d'un fichier :

Récupérable, un fichier est récupérable si en cas d'erreur il est possible de retrouver son dernier état cohérent.

Robuste, un fichier est dit robuste s'il est en mesure de subvenir à une éventuelle détérioration du support de stockage.

Un fichier récupérable n'est pas obligatoirement robuste et vice versa. Une journalisation de chaque opération en cours permet de basculer entre les différents états cohérents. La robustesse des fichiers est réalisée par la mise en place de techniques de réplication (*RAID*, *mirroring* ...).

La disponibilité des ressources est encore une autre propriété de la tolérance aux pannes (une information est disponible si quelque soit la panne, elle est accessible quand un nœud en a besoin). Le fait de ne pas accéder à une ressource par un client, peut être vu comme une erreur qu'il est nécessaire de prévoir (défaillance matérielle ou verrou bloquant selon la sémantique).

Enfin, l'ensemble des contraintes générées par l'utilisation des solutions de réplication est à prendre en compte. Du point de vue de l'utilisateur, cette gestion est totalement invisible. Cependant, il faut que chaque replicat d'un fichier sur le système soit cohérent selon la sémantique employée. La cohérence entre toutes les images physiques du fichier a un coût non négligeable. Il est parfois obligatoire d'utiliser des solutions effectuant les mises à jour sur un nombre limité d'images.

Une autre possibilité sont les DFS seulement utilisés en lecture (principalement les systèmes «pair à pair» et certains nouveaux modèles cf. paragraphe 2.4). Ce type d'architecture permet de distribuer les replicas lors de l'initialisation du système uniquement. Aucun erreur de cohérence pourra survenir.

1.6 Passage à l'échelle

La capacité d'un système à s'adapter à une forte progression de son taux de charge est appelé la *scalabilité*. Un DFS est rapidement sujet à être saturé par l'augmentation du nombre de clients. Le surcoût de trafic et l'analyse de chaque requête influent sur les capacités du CPU et du réseau sous-jacent. Ajouter de nouvelles ressources peut résoudre le problème mais génère parfois un surcoût additionnel, voire une remise en cause de la conception du système. Un DFS *scalable* doit pouvoir croître sans engendrer de tels phénomènes.

Dans un premier temps, afin d'éviter ces types de conflit, les architectures ont été bornées. Chaque grandeur décrivant l'architecture (débit réseau maximum, puissance d'analyse ...) était fixée à une constante qui limitait le nombre de ressources possibles. Par la suite, l'implémentation de serveurs sans ou avec états plus «intelligents» a permis de repousser certaines limites. Le serveur stocke les informations concernant chaque client pendant une période déterminée. Si au bout de celle-ci, le client n'a pas donné signe de son existence, les données sont effacées. Le serveur n'a plus à s'occuper de l'état de l'ensemble des clients (et est donc plus tolérant aux pannes).

Les principaux freins à la *scalabilité* sont :

Congestion et latence du réseau, même si de nombreux progrès ont été réalisés (gigaethernet, fibre optique ...), la plupart des *clusters* n'en sont pas encore équipés.

Type de sémantique, plus la sémantique est stricte, plus le passage à l'échelle est compliqué à mettre en œuvre et plus le surcoût de gestion est grand.

Puissance d'analyse, souvent limitée lorsqu'un nœud est «agressé» par une centaine de clients souhaitant être servis au même instant.

Les modèles fondés sur des approches centralisées ne sont pas scalables. L'idéal est un système symétrique où chaque composant du système a un rôle égal au sein du gestionnaire de fichiers. Cela devient possible au sein des nouvelles plates-formes où l'enjeu est justement d'utiliser les capacités de chaque nœud (les réseaux locaux constitués de console, TXs connectés à un unique gros serveur, ont été surpassés par les réseaux de PC). C'est cette problématique que nous abordons dans un contexte où nous souhaitons exploiter au mieux les ressources disques et les performances des réseaux sous-jacents.

Chapitre 2

Les solutions actuelles

Comme nous l'avons cité lors de l'introduction, la dispersion et le nombre des unités de traitements, au sein des architectures informatiques, suggère l'utilisation d'un système de fichiers distribué répondant à l'ensemble des contraintes et des avantages de ces nouveaux environnements (bande passante, réplication, accès partagé à l'information, gestion dynamique du matériel, allocation des ressources ou encore tolérance aux fautes...). Si certaines solutions présentent plusieurs points encourageants, elles sont, néanmoins, souvent spécifiques et complexes d'utilisation. Après avoir exposé la terminologie spécifique des systèmes de fichiers, nous allons aborder chacun de ces modèles afin de bien comprendre les problèmes qu'ils rencontrent et les solutions qu'ils proposent pour pallier à certaines de ces exigences.

2.1 Généralités

Les critères, que nous avons présenté lors du précédent chapitre, sont influencés par l'architecture du système, qui peut être soit fondée sur l'échange de messages entre deux machines (figure 2.1), soit sur la possibilité d'accéder directement à l'ensemble des disques disponibles (figure 2.2).

Chaque structure contient des subtilités permettant de faire jouer les différents facteurs. Les systèmes caractérisés par la transmission de messages sont, la plupart du temps, construits sur un modèle client/serveur. Les informations (les fichiers) sont conservées de façon pérenne sur le serveur. Le client possède, uniquement, en cache, une copie des données qui lui sont nécessaires. Cette solution présente l'avantage de pouvoir accéder, plus facilement à des ressources provenant d'un réseau local extérieur. En effet, pour des raisons de sécurité et de structure physique, il paraît, pour le moment, difficile d'utiliser un réseau en accès direct (Bus I/O) sur des disques entre plusieurs intranets. Pourtant, comme nous le verrons par la suite, cette approche offre d'excellent résultats au sein d'une grappe.

Nous avons, donc, rassemblé en deux sous chapitres, les systèmes de fichiers actuels répertoriés au sein des grappes. Une troisième rubrique évoquera très brièvement les divers protocoles de transfert usuels (HTTP, FTP ...) ou encore de protocoles dits pair à pair (gnutella, freenet ...). En effet, ces derniers pourraient s'avérer être d'éventuelles solutions au sein des grilles de calculs et peuvent être comparés comme des systèmes de fichier *Read Only*.

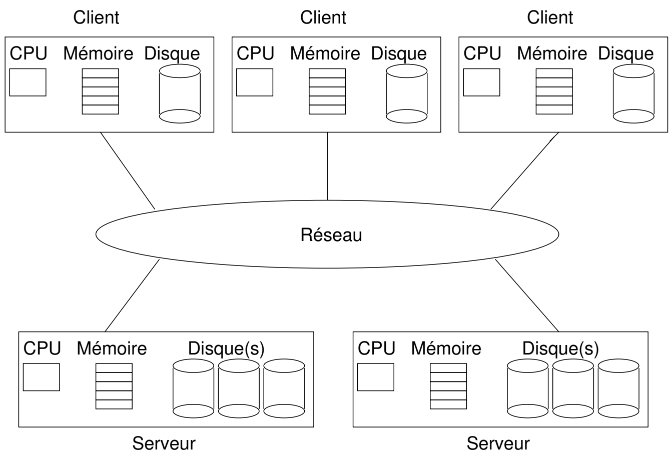

FIG. 2.1 – Modèle à messages

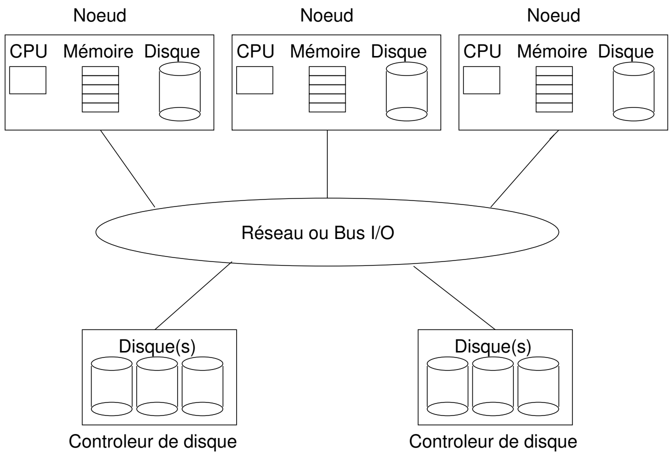

FIG. 2.2 – Modèle à accès direct

2.2 Modèle à base de messages

L'avantage de cette approche est son indépendance vis à vis de l'architecture matérielle sous-jacente. Les systèmes ont la même portabilité que le protocole qui est utilisé pour l'interconnexion des nœuds. Ainsi il serait tout à fait possible de partager des ressources entre plusieurs réseaux locaux. Néanmoins, pour des raisons de sécurité, cette fonctionnalité est utilisée dans des systèmes de fichiers hautement sécurisés.

2.2.1 NFS, *Network File System*

Le système de fichiers NFS est développé par Sun Microsystems depuis 1985. Fondé sur une approche client/serveur centralisé (figure 2.3), son but premier était l'implémentation d'un module indépendant de partage de fichiers offrant la transparence de nom tout en préservant une sémantique de type UNIX. Nous verrons par la suite que ce dernier point n'a jamais vraiment été atteint. La sémantique fournie par NFS est un point faible du système.

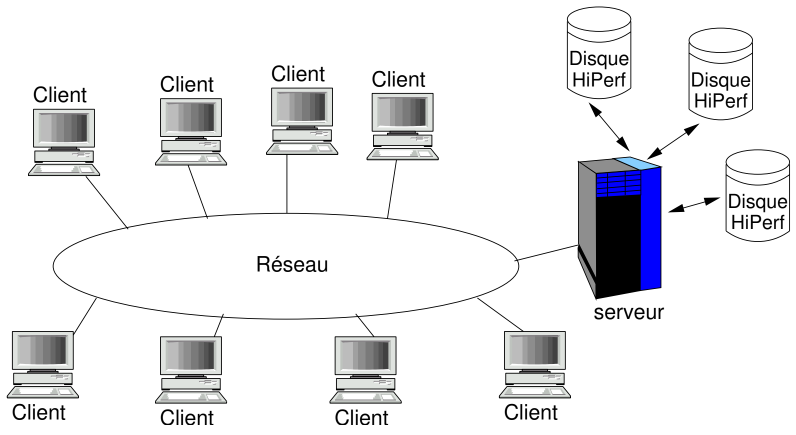

FIG. 2.3 – Architecture NFS

Les clients et le serveur communiquent via le réseau en utilisant plusieurs protocoles définis dans la norme NFS et spécifiés au sein du protocole SunRPC (*Remote Procedure Calls*). L'ensemble des services est fourni par différents processus démons, qui constituent le serveur ; ils sont accessibles par l'intermédiaire du protocole *mount* et du protocole *NFS*.

Le protocole *mount* : il est utilisé pour établir la connexion initiale entre le client (qui effectue l'appel) et le serveur (qui propose le service). Le serveur maintient une liste de l'ensemble des systèmes de fichiers qu'il exporte ainsi que les nœuds qui ont le droit de les monter. Lorsqu'un client tente de monter un système de fichier distant, le serveur après authentification, lui retourne un descripteur sur le système de fichier[1]. Cette identifiant permettra au serveur de résoudre les futures requêtes NFS envoyées par le client[2].

Le protocole *NFS* : il fournit un ensemble de procédures (RPC) pour réaliser des opérations à distance sur les ressources (recherche, manipulation des liens et des répertoires, gestion des attributs et enfin lecture et écriture sur des fichiers). Nous insistons sur le fait que toutes ces fonctionnalités sont accessibles uniquement après avoir invoqué la procédure de montage puisque c'est cette dernière qui retourne le descripteur nécessaire dans chaque requête. De plus, les opérations habituelles d'ouverture et de fermeture d'un fichier ne figurent pas au sein du protocole. Ces fonctions ne sont, en effet, d'aucun intérêt au sein d'un DFS sans état. Par ailleurs, l'utilisation des *RPCs* synchrones au sein du modèle NFS entraîne une mise en veille des clients à chaque appel. Les entrées/sorties sur le disque sont réellement exécutées avant de retourner la réponse au client. Le protocole NFS ne fournit pas de mécanisme permettant des accès concurrents. Un sous module (service de verrouillage) a été rajouté, par la suite, pour permettre d'y parvenir mais son utilisation doit être explicite.

[1] le terme de *partition* est utilisé

[2] D'un point de vue plus technique, l'opération de montage permet d'ajouter au sein de la VFS (*Virtual File System*) du client, la partition distante en y plaçant le descripteur correspondant. La VFS est l'image qui permet d'agréger la totalité des systèmes de fichiers présents au sein d'une unique machine (voir annexe A).

La totalité des messages transmis est au format UDP (bien qu'un mode TCP existe, le mode UDP est presque exclusivement utilisé, NFS étant un protocole sans état, l'utilisation d'un mode de communication «non-connecté» s'avère être naturel). Ce type d'échange offre une tolérance plus facile à implémenter face à d'éventuelles défaillances. Que ce soit au niveau du client, du serveur ou encore du réseau, les messages doivent seulement être renvoyés.

Néanmoins, comme nous l'avons précisé, la cohérence proposée par ce gestionnaire peut être qualifiée de relâchée puisqu'elle n'est fondée sur aucune sémantique définie. En effet, afin d'améliorer les performances, une gestion de cache (avec une granularité de 4Ko puis de 8Ko) a été mise en place au sein de l'interface coté client en s'appuyant sur une notion de temps (30 secondes pour les répertoires et 3 secondes pour les fichiers[3]). Un nouveau fichier créé sur un nœud ne peut être visible que 30 secondes plus tard sur les autres machines constituant la grappe.

Différentes versions ont été et sont implémentées, la première version de NFS est restée confinée au sein des laboratoires de SUN. La version 2 a été mise en place et livrée avec le système d'exploitation SunOS 2.01. Depuis cette date (courant 1985), NFSv2 est devenu un standard [RFC89] et est encore largement utilisée dans le monde UNIX malgré ses défauts (l'implémentation portable en mode utilisateur a aussi beaucoup contribué à sa diffusion). Actuellement, la version disponible en production est la version 3 et est normalisée par l'IETF [RFC95]. Cette version apporte principalement des optimisations au niveau des performances et conserve les grandes lignes du modèle précèdent. L'amélioration majeure est la possibilité de réaliser des écritures asynchrones. Les écritures synchrones augmentaient considérablement la latence, le client était mis en veille durant toute l'exécution de l'opération. Au sein de NFSv3, le client peut continuer son exécution dés que le serveur a pris acte de sa requête. L'opération d'écriture au sein du serveur peut être exécutée de manière à optimiser la charge ainsi que les accès disques. Par ailleurs, il est intéressant de préciser qu'il est possible de stocker des fichiers de grande taille (> 2Go) ce qui n'était pas le cas dans les versions précédentes (cette fonctionnalité est proposée pour tous les noyaux linux supérieur à 2.4 et sur les nouvelles architectures 64 bits).

Ce système de fichier convient parfaitement à de petits *clusters* ou à des réseaux locaux sécurisés de taille restreint (jusqu'à 25 machines) mais les performances s'effondrent très vites si les applications sont gourmandes en E/S et ceci, principalement, à cause de la nature centralisée du système. NFS est d'ailleurs plutôt considéré comme un service de réseau qu'un composant d'un système réellement distribué. Néanmoins, son implémentation simplifiée permet au système d'offrir une grande stabilité.

Sun a réalisé une version 4 du protocole [RFC00], [PSB$^+$00] courant 2000. Ce nouveau système essaye de tenir compte des problèmes actuels des *clusters* (en autre la réplication/migration des données ou encore le fait de pouvoir partager des informations entre des entités de l'internet). Pour ce faire, un remaniement profond de l'architecture a été effectué.

Les principaux changements au sein de NFSv4 sont :

Connexions *TCP*, cet aspect est nécessaire car l'UDP ne fournit pas le mécanisme de contrôle de flux requis pour un trafic performant sur l'Internet.

Sécurité renforcée, plusieurs nouveaux mécanismes d'authentification ont été rajoutés au sein des RPCs.

[3] Ces paramètres sont réglables sur chaque client

Système à état, le protocole ne sera plus sans état et ce afin d'avoir un unique protocole pour accéder aux différents services (monter les partitions, gérer les verrous,...). Cette technique permettra d'éliminer les divers modules annexes du systèmes.

Protocole unique, l'ensemble des services est intégré au sein d'un seul protocole.

Néanmoins l'implémentation sous Linux, développée fin 2000, n'a pas été maintenue et le modèle n'a pas encore été unanimement accepté. les solutions de stockages construites sur NFS et proposées par les industriels utilisent toujours la version 3 du gestionnaire.

Plusieurs projets sont des variantes de l'architecture NFS :

ClusterNFS, il fournit un niveau d'abstraction[4] permettant de spécialiser les partitions exportées par rapport à chaque nœud (lorsqu'une machine, par exemple «Cluster1», accédera au fichier `/etc/hostname.conf`, le serveur traitera une requête de la forme `/etc/hostname.conf.Cluster1`).

WebNFS, un accès fiable avec une indépendance de location est proposé sur intranet comme sur internet. Développé par Sun en 1997, ce système permet de partager des ressources au travers de l'internet grâce à la mise à disposition d'un fichier contenant les correspondances descripteur de fichier : partition «montable» (suppression du protocole mount). Néanmoins, ce système n'a pas eu l'essor escompté.

2.2.2 De SPRITE à xFS

SPRITE

SPRITE est un système d'exploitation distribué pour un réseau de machine *RISCs* (processeur Motorola). Développé, dans le cadre du projet SPUR (Université de Californie à Berkeley), de 1985 à 1993, il avait pour but la conception et la construction d'un *cluster* performant [OCD$^+$88]. Basé sur un noyau multithreadé, SPRITE a été développé dans l'idée que la prochaine génération des postes de travail seraient des machines puissantes constituées d'une grande capacité de mémoire vive (à l'époque, 32Mo de RAM, aujourd'hui, des systèmes possédant 256 Mo ou 512 Mo sont plus que courants).

Le système de fichiers, inclus dans SPRITE, fournit un espace de nommage unique (transparence de situation) et une sémantique de type UNIX. Distribué au travers de plusieurs serveurs et clients, il intègre une gestion de cache maintenue en cohérence par le serveur (serveur avec état). Les divers appels systèmes (création, lecture, écriture, ...) sont construits, comme NFS, sur le protocole RPC.

L'arborescence de fichiers unique, proposée à l'utilisateur, est partitionné de manière transparente en *domaines*. Chaque nœud, composé d'un espace de stockage à long terme, peut contenir un ou plusieurs domaines (et peut se comporter, par conséquent, comme client d'une autre machine mais aussi éventuellement serveur des entités distantes).

Une table de correspondances (appelé table des *préfixes*) entre les différents domaines et leur situation physique sur la grappe est maintenue dynamiquement au sein

[4] `http://clusternfs.sourceforge.net/`

de chaque nœud. Ainsi, lorsqu'un client tente d'accéder à un fichier, une recherche au sein de cette table est effectuée afin de récupérer l'adresse du serveur qui a la charge du domaine dans lequel la ressource figure. Si la table ne contient pas l'entrée correspondante, une requête est diffusée vers toutes les machines de l'architecture pour connaître le propriétaire de la ressource. Cette technique permet de mettre à jour dynamiquement la table et surtout de pouvoir fournir une possibilité de migration des données[5] (même si cette dernière n'a pas été mise en œuvre au sein du modèle). Par ailleurs, afin d'éviter les contraintes de gestion d'un espace de nom unique (les répertoires `/dev` ou certains fichiers de configurations spécifiques à chaque composant de la grappe), il est possible de «privatiser» certaines ressources au sein des tables de *préfixes*.

Par ailleurs, dans le but d'offrir la sémantique UNIX, une gestion de cache scindée en deux sous modules a été implémentée au sein du modèle. Le premier se rapproche assez du modèle proposée par NFS, fondé sur un notion de temps, il permet de réaliser des écritures retardées et ainsi optimiser les accès pour les fichiers temporaires (cf paragraphe 1.3). De plus, un numéro de version est associé à chaque image disponible ; dès qu'une ressource est ouverte en écriture, ce numéro est incrémenté. Ainsi, lorsqu'un client accède à l'information, il compare le numéro de version présent dans son cache avec le numéro de version présent sur le serveur ; si ces derniers sont identiques, il n'est pas nécessaire de fournir à nouveau les données. Néanmoins, afin d'éviter les problèmes d'incohérence entre les multiples images disponibles sur l'architecture (le serveur ne contient pas forcément la dernière version, ceci étant dû au concept des écritures retardées), une méthode d'invalidation des caches (le second sous-module) a été intégrée au sein de chaque serveur. Lorsque un nœud accèdent en écriture au sein d'une ressource et que d'autres clients souhaitent y accéder en lecture, le serveur force «l'écrivain» à lui retourner les changements. De même, lorsque plusieurs nœuds partagent une ressource en lecture et qu'un autre aspire à y effectuer des modifications, le serveur invalide le cache des «lecteurs». Le serveur possède donc des informations sur l'ensemble des connexions en cours (serveur avec état). C'est le coût à payer (émissions de nombreux messages, problèmes liés à la tolérance aux fautes) pour pouvoir proposer une sémantique de ce type.

La conception du système fondée sur la gestion de l'ensemble des données en mémoire vive permet d'obtenir des résultats intéressants. Cependant, la complexité d'implémenter au sein du modèle (serveur avec état) un DFS tolérant aux fautes apparaît être une réelle contrainte. Mais encore et surtout, la technique de diffusion pour la mise à jour des tables de *préfixes* n'offre malheureusement pas un grand coefficient de scalabilité (la diffusion à outrance comporte un surcoût au niveau réseau non acceptable au sein des nouvelles architectures).

LFS

Au début des années 1990, une seconde version du système de fichier SPRITE fut produite. Elle est fondée sur un nouveau placement physique des données sur les disques. Appelée *LFS*[6] [OD92] [RO91], elle a été développée à partir de l'idée que les performances d'un système de fichier sont forcément dépendantes du matériel sous-jacent. Un disque est évalué par les deux critères suivants : la bande passante et le temps d'accès. Si la bande passante peut croître en agrégeant plusieurs disques (utilisation des

[5]Nous rappelons que la migration des données est importante au sein des architectures qui se veulent *scalables* puisqu'elle permet de réguler la charge entre les différents nœuds.

[6]*Log-structured File System*.

RAIDs), le temps d'accès est limité physiquement et diminue proportionnellement aux avancées technologiques. Ainsi, le but de LFS a été de proposer une solution permettant d'éviter, au maximum, les diverses opérations «mécaniques» sur les disques.

Suite aux précédents travaux sur le DFS SPRITE, il a été observé que le trafic réel des données est dominé par les écritures (nous rappelons que SPRITE est fondée sur le fait de disposer d'un cache imposant qui par conséquent optimise les requêtes de lecture). Afin de bien comprendre ce qui différencie la structure de LFS des systèmes de fichiers habituels , nous allons étudier ce système dans un environnement local (une seule unité de stockage). Toute nouvelle information sur le disque est disposée de manière séquentielle (comparaison avec un tampon circulaire où les données sont inscrites à la suite). Cette approche augmente considérablement les performances en écriture en éliminant les déplacements de la tête de lecture du disque. A la différence d'un système de fichiers habituel [Bac86], toute modification sur un fichier (changements sur les données ainsi que sur les entêtes[7] de la ressource) est stockée de manière contiguë et avec au pire un déplacement (l'écriture s'effectue à la suite des dernières informations enregistrées). Certains systèmes proposaient et proposent encore cette notion de journalisation au sein de la gestion de cache (cf. paragraphe 1.5) mais LFS fut le premier système à enregistrer de manière pérenne les informations sous un tel format.

Les schémas suivants (figure 2.4) illustrent très bien ce nouveau concept de positionnement des données au sein d'un disque.

La figure (a) montre une implémentation traditionnelle des données au sein d'un disque (d'un côté, les méta-données[8] et de l'autre les informations réelles). Lors d'une modification, il est impératif d'accéder au bloc de méta concerné afin par exemple d'y affecter un pointeur vers un nouveau bloc. La figure (b) montre une première évolution vers les structures de fichiers s'appuyant sur le concept de journalisation. Les nouvelles données sont stockées à la suite des précédentes (la fragmentation du disque est ainsi temporairement évitée). Nous verrons, par la suite, le moyen utilisé pour mettre à jour les données sur le disque qui se comporte comme un tampon circulaire. La figure (c) offre un niveau d'abstraction permettant d'éliminer la notion de deux entités (à savoir les méta et les données) a des endroits spécifiques au sein du disque. Pour cela, un bloc de super méta est utilisé : il contient les divers pointeurs sur les différents inodes. Ces derniers peuvent alors être journalisés parmi les blocs de données. La taille du super bloc de méta doit être inférieur à celle du cache afin d'éviter les accès sur disque pour le modifier. Dans le cas contraire, aucun gain n'est obtenu puisqu'il est toujours nécessaire d'effectuer deux déplacements. Le dernier schéma présente la structure implantée au sein de LFS. Il s'agit d'écrire occasionnellement le bloc de super méta toujours de façon journalisée. De cette manière, après un éventuel dysfonctionnement obligeant le redémarrage de la machine, il suffit de récupérer le dernier bloc de super-méta stocké pour revenir dans un état cohérent du système de fichiers. Il est possible, d'examiner les données comprises entre la dernière écriture du super bloc et la fin du tampon circulaire afin de reconstruire au mieux les informations.

La principale difficulté, de ce modèle, est la réorganisation des blocs lorsque le pointeur d'écriture atteint la fin du disque. Toute prochaine opération d'écriture va s'effectuer en tête du support de stockage (tampon circulaire) et par conséquent écraser des données potentiellement valides. En effet, il apparaît, d'après l'étude menée par

[7] communément appelé inode ou méta-données.

[8] Les méta-données contiennent les informations relatives aux droits d'accès, les différentes estampilles de temps (dernier accès, modification) et enfin les références vers les blocs de données.

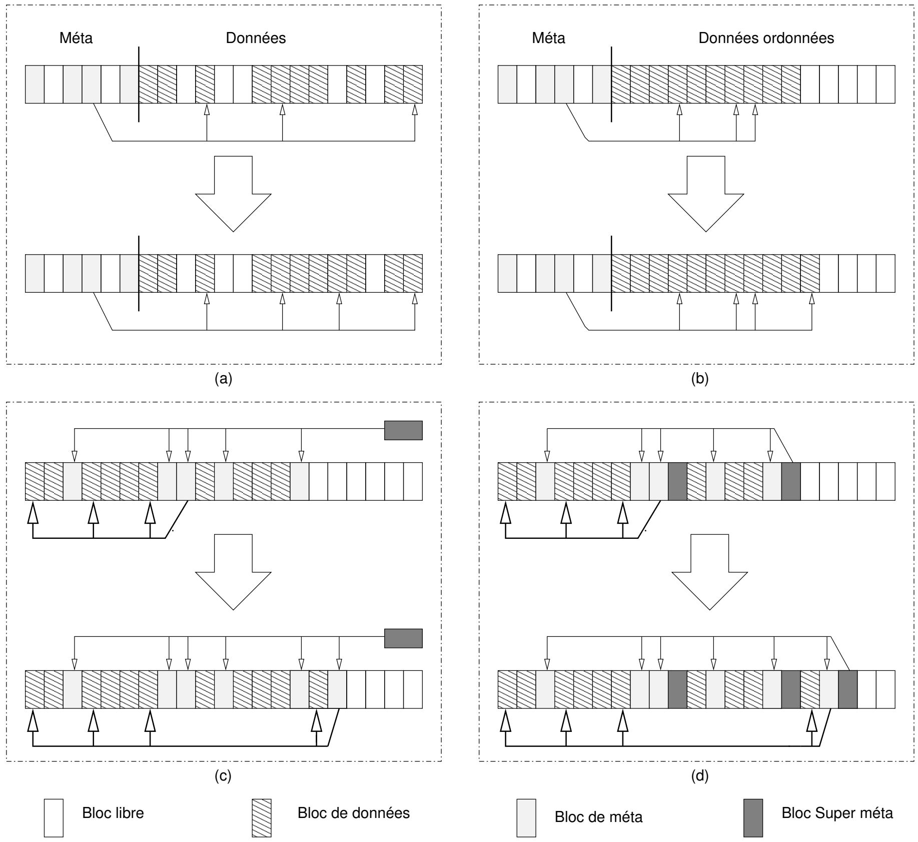

FIG. 2.4 – Du système de fichiers traditionnel au modèle LFS

l'équipe de LFS, que 90% de l'espace disque doit être inutile (donc libre) lorsque le pointeur d'écriture a parcouru la totalité du disque (le disque est totalement défragmenté). Afin d'éviter ce problème, un mécanisme de nettoyage est utilisé. Le disque est découpé en plusieurs segments et avant toute écriture sur un nouveau segment, le précédent est mis à jour (les données encore valides sont transférées sur le segment suivant). Le disque contient au pire un segment fragmenté. Plusieurs politiques ont été implémentées [RO91] dans le but de minimiser le surcoût engendré par ce service.

Ce type de structure s'avère être performant et est intégré aujourd'hui dans de nombreux systèmes de fichiers locaux[9] [SBMS93]. Au sein du modèle SPRITE, la gestion de cache s'avère excellente puisqu'elle permet de récupérer une quantité importante de données afin de répondre rapidement au lecture et de réaliser des écritures (chacune étant composée de nombreux blocs) beaucoup plus performantes que les systèmes standards. Malheureusement comme nous le verrons dans le DFS suivant, la distribution du bloc de Super méta au sein d'une architecture de type *cluster* s'avère complexe et coûteuse. Ce système de fichier a surtout révélé et permis d'étudier une nouvelle méthode de positionnement des données au sein d'un disque. Diverses améliorations [MRC+97] ont été apportées au niveau des techniques utilisées pour le «nettoyage» du disque et

[9] `http://collective.cpoint.net/lfs/`

des méthodes de lecture.

Zebra

Courant 1992, le système de fichier Zebra est apparu au sein du projet SPRITE. Fondé sur la technologie de LFS, il a été conçu dans l'optique d'améliorer les débits de transfert entre les diverses machines en agrégeant plusieurs flux. Les fichiers sont répartis par paquets sur plusieurs serveurs dédiés (appelé serveur de stockage) [10]. La répartition des données sur plusieurs machines permet de distribuer la charge totale entre les serveurs et de cette manière réduire la probabilité de saturer un nœud.

Par ailleurs, Zebra utilise un serveur de stockage afin de contrôler la parité du *stripe* (cf. figure 2.5). Cela permet d'augmenter la tolérance aux pannes (une ressource est toujours accessible même avec un disque défaillant) sans pour autant générer un surcoût intolérable (souvent engendré par la technique de réplication).

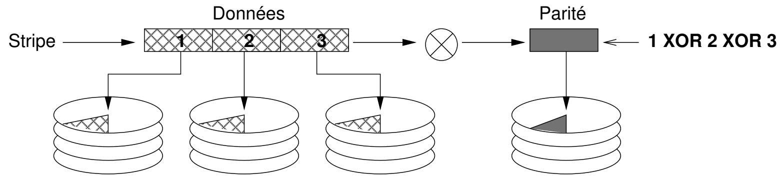

FIG. 2.5 – Méthode de *striping* avec contrôle de parité

La technique du *striping* comporte certaines subtilités : si la répartition est effectuée par fichier (chaque serveur est chargé de ressources spécifiques), le taux de charge est théoriquement distribué entre l'ensemble des serveurs mais certains nœuds seront beaucoup plus «agressés» (par exemple, le répertoire `/usr` pourrait être attitré à un serveur qui se révélerait alors être un point de congestion). Lors d'une répartition à un grain plus fin (un fichier est répartie sur plusieurs nœuds), de nouveaux paramètres apparaissent. Si le fichier est plus petit que le grain défini, nous retombons dans le cas précédent et si la taille des paquets choisie est trop faible alors la latence du réseau et des disques s'avère énormément coûteuse pour la quantité de données récupérée. Afin de remédier à ces problèmes, une répartition par client a été implémenté au sein du DFS Zebra. Utilisant la technique de structuration des données proposée par LFS, chaque client organise ses informations (nouveaux fichiers, modifications ...) de manière journalisée au sein d'un tampon (appelé *log*) qui, une fois plein, est réparti sur les serveurs de stockage dédiés.

Nous allons étudier l'architecture du système de fichiers Zebra (figure 2.6) afin de visualiser les techniques mise en œuvre pour le partage des ressources. Chaque client écrivant ses *logs* indépendamment des autres entités, une politique d'arbitrage est par conséquent impérative. Pour cela, Zebra a implémenté une gestion centralisé de tous les *logs* qui transitent sur le réseau.

Les clients : pour les opérations de lecture, ils interrogent le gestionnaire de fichiers qui détermine sur quel fragment se situe les données souhaitées ; le client n'a plus qu'à formuler sa requête au serveur de stockage. Les écritures suivent le

[10]Technique de *striping*.

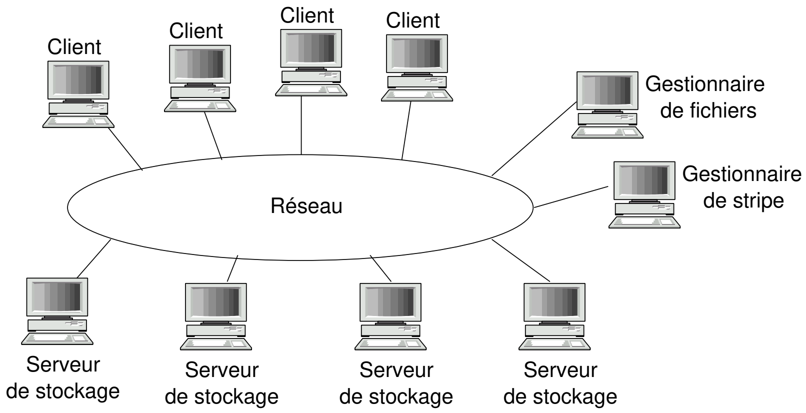

FIG. 2.6 – Architecture du système de fichier Zebra

modèle LFS : un «journal» des modifications effectuées, composé des données et de méta-informations spécifiques au DFS Zebra (les *Deltas*) est fragmenté puis envoyé au serveur de stockage. Les *deltas* sont transmis, en même temps, au serveur de fichiers et de *stripes* afin de maintenir en cohérence le système et d'éviter ainsi à ces services d'accéder au serveur de stockage.

Le gestionnaire de fichier : il permet, d'une part, de retrouver où sont situés les données d'un fichier sur les différents serveurs de stockage et d'autre part de maintenir la cohérence entre les différents caches des clients. A chaque fichier créé par un client correspond un méta-fichier stocké sur le gestionnaire de fichier. Les «données» de ce méta-fichier sont des références vers les différents blocs enregistrés sur les serveurs de stockage. La cohérence du cache est réalisée par l'accès à ces «méta-fichiers» et a été implémentée d'après le modèle inclus dans SPRITE.

Les serveurs de stockage : ce sont des entités «simples» qui gèrent des blocs (ou fragments) de taille fixe (512Ko). Les services proposés sont : le stockage, la récupération, l'identification, la suppression de fragments. Une opération de mise à jour a été ajoutée afin de pouvoir compléter des fragments non plein (lorsqu'un client force l'écriture d'un tampon qui n'est pas encore totalement rempli, il est intéressant qu'il puisse utiliser la place libre du fragment par la suite).

Le gestionnaire de *stripes* : Ce service a à sa charge l'organisation des segments au sein des divers serveurs de stockage (défragmentation des disques afin d'obtenir des segments libres, cf. LFS). A chaque *stripe* correspond un fichier Zebra (*stripe status file*) qui renseigne le gestionnaire sur le nombre de données encore valide au sein du *stripe*. Ces informations sont mises à jour grâce aux *deltas* retournés par les clients.

Les performances et la tolérance aux fautes du gestionnaire de fichiers sont un souci. Ressource centralisée, elle rend complexe voire impossible le passage à l'échelle du système. Par ailleurs, les méta-informations sont stockées sur un disque souvent propre au gestionnaire de fichiers. Il serait intéressant d'enregistrer ces informations sur les différents serveurs de stockage et permettre de cette manière à un autre nœud de reproposer rapidement ce service. En cas de dysfonctionnement, imposant l'arrêt du serveur, le cache peut être recréé selon la méthode décrite par LFS.

La disponibilité des fichiers fournie par la technique de parité permet d'accéder à une ressource même si une unité de stockage venait à tomber en panne. Cependant, là encore, la gestion de l'intégrité des *stripes* s'avèrent être un problème : si une défaillance survient lors d'une écriture (seule une partie du *stripe* a été sauvegardée) et qu'il manque plus d'un fragment, le *stripe* est alors définitivement perdu. Un protocole permettant de rendre cette opération atomique est plus que requis.

Un point fort du système est la possibilité d'ajouter un serveur de stockage à la volée. Il suffit alors d'avertir l'ensemble des clients et les deux gestionnaires pour continuer à utiliser le système (les anciennes informations n'ont pas à être de nouveau distribuées). Le gestionnaire de *stripes* va peu à peu effectuer ce travail.

xFS

Depuis 1993, le projet *NOW*[11] a remplacé SPRITE. Un nouveau DFS, fondée sur LFS et sur Zebra a fait son apparition. Une première version était dirigée vers la hiérarchisation des serveurs de méta-données [WA93] afin d'améliorer les performances et supprimer le goulot d'étranglement lié à l'approche centralisé de Zebra. Courant 1995, une solution [ADN+95] dite «sans serveur[12]» a suivi la précédente version. Fondée sur l'idée que l'ensemble des services fournit par un système de fichiers doit être accessible sur tous les nœuds, chaque machine du *cluster* peut stocker, contrôler, effectuer du cache et fournir les ressources au sein de xFS. Pour permettre cela, xFS utilise un espace de nommage unique et offre une indépendance à la mobilité (cf paragraphe 1.2). S'appuyant sur les point forts des implémentations des DFS précédents, il enregistre les informations selon une approche journalisée (à la LFS) et fournit la technique du *striping*, augmentée de la parité, pour améliorer les débits de transferts, la disponibilité et obtenir une meilleure tolérance aux pannes.

D'un point de vue architecture matérielle, xFS a été développé pour s'exécuter sur des machines inter-connectées par un réseau rapide. Ce critère est prépondérant dans la conception de xFS. Fondé sur les travaux de SPRITE (utilisation de nœuds performant à grande capacité de mémoire), xFS accède en priorité aux caches distants des autres machines. Cette gestion de cache «coopérative» découle du fait, qu'à l'époque, un accès disque pouvait s'avérer plus coûteux, en terme de latence, qu'une opération de lecture sur une mémoire distante (ce qui est toujours vrai sur un *cluster* relié par un réseau Gigabit).

La gestion de la cohérence est toujours fondée sur celle proposée au sein de SPRITE ; le grain a cependant été modifié. La cohérence s'effectue au sein d'un bloc système et non plus au niveau du fichier. Cela permet d'obtenir une meilleur disponibilité lors du partage de ressources.

xFS est basé sur l'architecture de Zebra, la principale différence est la distribution de gestionnaire de fichier et du gestionnaire de nettoyage sur plusieurs nœuds (voir la totalité). Chaque entité de la grappe peut agir comme un client, un gestionnaire de stockage, de fichiers (méta-informations) et de nettoyage (remise en place de segments inutilisées, défragmentation). Tout goulot d'étranglement est ainsi évité. L'implantation des clients et des serveurs de stockage est peu différente du modèle Zebra. Par

[11] Network of Workstations
[12] *Serverless Network File Systems*

conséquent, nous allons nous intéresser à l'implémentation de la distribution du gestionnaire de méta-informations, dans un premier temps ; nous nous attarderons, par la suite, à celle du service de nettoyage des disques. Un gestionnaire de fichier gère un sous groupe d'identifiants (un ensemble d'inodes est réparti sur chaque gestionnaire). A la différence de SPRITE, où la table des *préfixes* était dynamique, la table de répartition des identifiants est statique et est répliquée sur chaque gestionnaire. De cette manière, lorsqu'un client souhaite accéder à une ressource, il lui suffit de regarder au sein de la table pour savoir quel est le gestionnaire qui en a la charge. Dans le but d'optimiser les performances, un fichier créé par un client se voit attribuer un identifiant dépendant du gestionnaire de méta local. Cette approche, référencée sous le concept du «premier écrivain», permet de limiter le nombre de messages transitant sur le réseau. Néanmoins, comme nous l'avons déjà cité, la répartition des informations par fichiers peut se révéler dangereuse dans un système qui se veut *scalable* (cf. paragraphe 2.2.2). L'équipe qui a développé xFS a soulevé ce dernier point et émis diverses solutions permettant de rééquilibrer les charges : ré-allocation dynamique d'une inode à un autre gestionnaire ou affectation, dès le départ, de plusieurs identifiants distribués sur les gestionnaires (chacun pointant vers une portion spécifique du fichier).

Le service de nettoyage de disque a pour tâches la sauvegarde de l'état de chaque segment (combien de blocs sont devenus inutiles au sein du segment ...), la comptabilité des zones libres et occupées sur chaque disque et enfin la remise en état de ces derniers (copier les données encore valides au sein d'un segment vers un nouveau afin de libérer celui-ci en vue d'une prochaine utilisation). Dans le cadre de la distribution de ce service, chacune des tâches a été assignée à une unité de l'architecture. Ainsi, chacun des clients maintient à jour les informations relatives aux segments qu'il a écrit (ces renseignements sont stockés au sein de fichiers xFS, nommés *s-files*). Cette solution permet d'éviter, là encore, l'utilisation inutile du réseau : des messages seront émis uniquement lorsque des clients effectueront des modifications sur des blocs de données stockés au sein de segments ne leur appartenant pas. Pour coordonner l'ensemble des «nettoyeurs», un gestionnaire élu *leader*, assure la comptabilité et attribue, lorsque cela est nécessaire, un nombre différents de segments (en fonction de la charge) à chaque nœud pour une mise à jour. La technique de cohérence entre les clients et les gestionnaires est la même que dans Zebra (utilisation des *deltas*).

Toujours dans le but d'améliorer le DFS, les serveurs de stockages sont répartis en sous groupe de même taille (*stripe group*) et contenant tous un serveur de parité. L'écriture simultanée d'un *stripe* par plusieurs clients est alors parallélisée sur l'ensemble des groupes. Par ailleurs, en cas de dysfonctionnement d'un serveur de stockage au sein d'un groupe, seul l'accès en lecture est autorisé (le groupe est alors dit *obsolète*), toute écriture est réalisée sur les groupes distants (qualifiés d'*actuels*). Cette approche élimine le problème qui apparaissait dans Zebra : à savoir la réalisation du calcul de l'ensemble des données manquantes à partir des blocs de parités lors du retour à la normal du serveur défaillant.

Les techniques de tolérance aux fautes s'appuient sur les solutions présentées par LFS et Zebra. Cependant la chronologie suivante est à respecter lors d'une remise en marche du système : gestionnaire de stockage, de méta (ou de fichiers) et enfin service de nettoyage. Cette enchaînement est dû à l'imbrication des services entre eux. Le serveur de méta requiert le serveur de stockage pour reconstruire son cache. De même, le serveur en charge de l'état des segments utilise les informations délivrées par le service de fichier pour accéder aux *s-files*.

La sécurité est le point qui fait, malheureusement, défaut au modèle. En effet, xFS est seulement approprié aux environnements sécurisés. L'accès par les clients aux divers caches peut se révéler désastreux si un de ces derniers est corrompu (un *log* malformé et sauvegardé sur le disque pourra empêcher un retour à un état cohérent lors d'un dysfonctionnement). Néanmoins, xFS propose une solution type «boîte noire» par l'utilisation de clients différents, par exemple NFS (figure 2.7). Chaque client NFS monte une partition via un serveur NFS exportant l'espace d'un client xFS. Les performances sont quelque peu altérées mais le système est sécurisé.

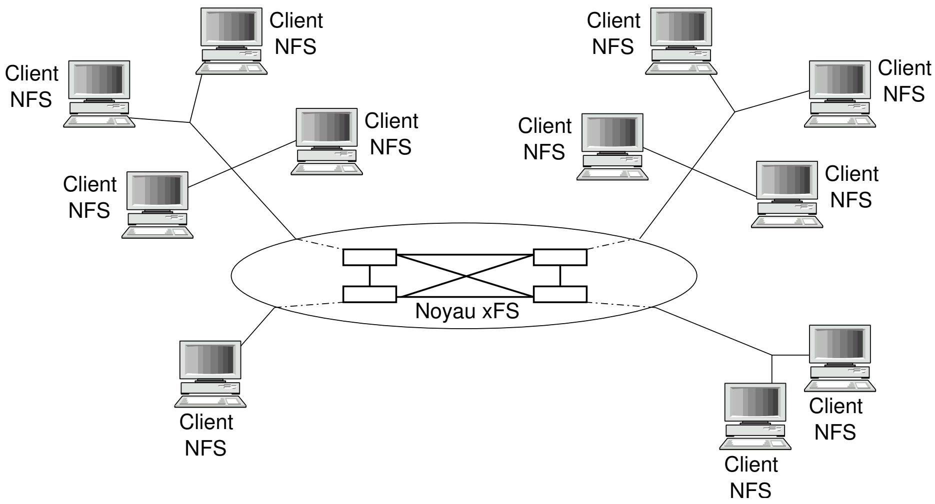

FIG. 2.7 – Utilisation d'une «boite noire» xFS par des clients NFS

Une dernière version[13] implémentée en mode noyau sur des machines Solaris a été mis à la disposition en 1997. Le projet NOW s'est terminé courant 1998.

2.2.3 AFS et ses descendants

AFS, *Andrew File System*

Il est impossible de parler des systèmes de fichiers distribués sans présenter ce que nous appellerons la «famille» AFS. En effet, ce système est un des projets les plus anciens parmi les DFS. Développé depuis 1983, à l'université de Carneggie Mellon (CMU), il a eu pour premier objectif de résoudre les problèmes de passage à l'échelle (jusqu'à 5000 nœuds à l'époque).

Le modèle est fondé sur une architecture clients/serveur structurée en sous *clusters* inter-connectés par un routeur à un réseau fédérateur (figure 2.8). Chaque sous groupe est constitué d'un nombre de poste «clients» et d'un unique serveur (cette unité est nommée *Vice* au sein du système). La charge est, de cette manière, répartie sur une collection de serveurs dédiés fournissant chacun le même ensemble de services à un groupe «restreint» de clients.

[13] `http://now.cs.berkeley.edu/Xfs/release/`

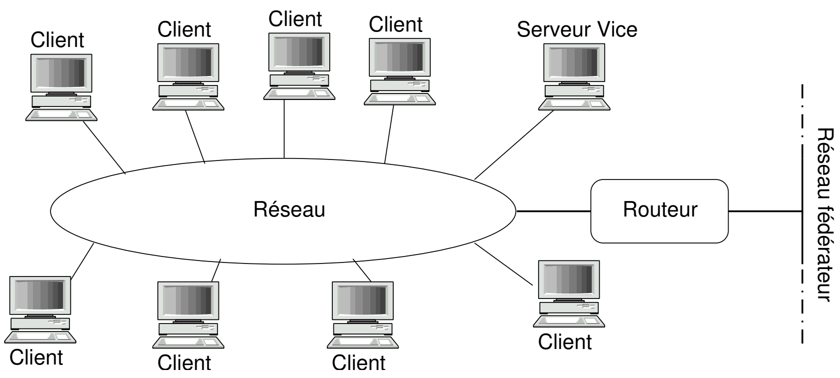

FIG. 2.8 – Architecture du système de fichier AFS

Chaque client dispose d'une arborescence partitionnée en deux parties distinctes : un espace de nom local et un espace partagé (sous les systèmes d'exploitation UNIX, la racine `/` correspond à l'espace local alors que le répertoire `/afs` permet d'accéder aux ressources globales et partagées). Un protocole simple construit sur les RPCs permet de récupérer la totalité d'un fichier. En effet, la principale caractéristique de AFS est une gestion de cache effectuée sur la globalité d'une ressource. Ainsi lors de l'accès à une information, la totalité des données est récupérée par une unique requête ; toute opération de lecture ou d'écriture peut être effectuée directement sur la cache sans pour autant émettre des messages vers le serveur.

Une ressource est associée à un identifiant unique (*fid*) au sein de l'espace partagé. Cette identifiant permet d'offrir une indépendance à la mobilité. Les informations permettant de retrouver l'emplacement physique d'un fichier sont enregistrées dans une base de données qui est répliquée sur l'ensemble des serveurs *Vices*. Lorsqu'un client tente d'accéder à une ressource de l'espace commun : l'appel système sous-jacent est intercepté et est transmis à un service utilisateur nommé*Venus*. Ce processus est le seul qui interagit avec le serveur, il émet une requête au démon *Vice* afin de récupérer la totalité du fichier. Une fois que la ressource est «cachée» (les données sont copiées sur le disque local), le client affecte un verrou virtuel sur le fichier (appelé *callback*[14]). La ressource est ainsi valide tant que le serveur n'a pas retiré ce *callback*. Le serveur est, par conséquent, avec état puisqu'il maintient la cohérence des données entre les différents clients. Lorsqu'un client récupère un fichier pour une écriture, l'ensemble des *callbacks* est retiré, le serveur transmet des messages d'invalidations à chaque clients concernés. Enfin, lors de la fermeture d'une ressource par un client, le processus *Venus* intercepte, là encore, l'appel système, vérifie si des modifications ont été réalisées et les transmet au serveur le cas échéant (sémantique de session). Cette solution de *callbacks* permet de minimiser le nombre de messages échangés entre les clients et les serveurs ; en effet, il est possible de renouveler l'accès à un fichier présent dans le cache sans en avertir le serveur tant que le verrou sur la ressource est valide.

Lorsqu'un équilibrage de charge est nécessaire pour soulager un serveur ou pour obtenir de meilleures performances (les requêtes arrivent principalement d'un sous *cluster* différent), les ressources en cause sont migrées simplement. Pendant la période

[14]Cette terminologie est un peu confuse, *callback* fait référence au nom plutôt qu'au verbe. C'est une abréviation pour *callback promise*

de transition, le serveur garde une copie afin de continuer à mettre à jour le fichier, les modifications seront envoyées par la suite. Toute opération d'équilibrage est atomique afin de ne pas corrompre le système en cas de défaillance (toute migration n'est validée qu'une fois l'opération réellement terminée). Une technique de réplication est proposée pour soulager les serveurs, néanmoins, une image est en mode lecture/écriture alors que les autres sont uniquement accessibles en lecture.

La sécurité au sein du modèle s'est révélée être un point important. Tout d'abord, aucun processus utilisateur ne peut et n'est autorisé à être exécuté sur un serveur *Vice*. De plus, un protocole d'authentification a été mis en place entre chaque client et le serveur. Mais encore, l'ensemble des messages échangés entre les diverses entités composant le système est crypté. Enfin, l'ensemble des informations sur les clients est stocké au sein d'une base de données protégée.

La prise en charge dynamique de clients au sein du modèle (possibilité d'ajouter ou de retirer des nœuds clients) est un avantage considérable qui est apparu dès la version 2. Plusieurs versions apportant des gains de performances se sont succédées. Actuellement, IBM propose une version OpenSource[15] d'AFS depuis la fin de l'année 2000 et de nombreux organismes utilisent encore ce logiciel (notamment CMU et CERN). Cette version est héritée de la version 3 qui a apporté un support pour un cache par morceaux de 64Ko. En effet, le partage de fichiers de taille imposante (plusieurs Mo) ne permettait pas d'obtenir des performances correctes.

CODA, *Constant Data Availability*

A partir de 1987, parallèlement à AFS version 2, le développement du projet CODA [SKK+90, NS94] a débuté et se poursuit depuis lors au CMU. Même si ce système comporte de nombreux points communs avec AFS (espace de nommage, sécurité), il a pour principal objectif, comme son nom l'indique, de fournir un taux de disponibilité maximal. Le système garantit un accès aux ressources dans tous les cas (défaillance d'un serveur, perte du réseau ...).

Pour y parvenir, le DFS CODA utilise les deux techniques suivantes :

Réplication «optimiste» : plusieurs images d'une même ressource, toutes accessibles en lecture/écriture, apparaissent sur différents serveurs. A l'opposé, une réplication «pessimiste» est employée lorsque la sémantique souhaitée doit être stricte (une unique image en mode lecture/écriture comme dans AFS) et éviter ainsi les conflits. L'approche «optimiste» vient du fait que les clients partagent rarement des ressources en écriture. Le cas échéant une détection d'incohérence (vecteur d'informations) est intégrée au sein du protocole et l'erreur peut être corrigée automatiquement ou manuellement par la suite.

Accès en mode déconnecté : cette méthode offre la possibilité de travailler sur une ressource même si aucun serveur n'est accessible. Parmi ce mode, deux approches se distinguent : le mode involontaire (défaillance matérielle) et la déconnexion volontaire (l'utilisateur travaille sur une machine portable et se retire simplement du réseau). Dans cette seconde approche, une nouvelle contrainte apparaît, en effet, aucune indication ne permet de prévoir à quel instant la machine réintégrera le *cluster*. Dans les deux cas, ce mode est réalisable grâce à la gestion du cache à gros grain (récupération de la totalité de la ressource). Durant toute la période «non connectée» l'utilisateur effectuera les modifications au

[15] `http://www.openafs.org/`

sein de son cache. Afin d'éviter la pénurie lors de l'accès à une ressource qui n'a pas été récupérée avant la déconnexion, l'utilisateur peut paramétrer son cache et préciser les informations qu'il souhaite avoir en permanence avec lui.

Pour permettre d'obtenir une certaine cohérence à court et à long terme tout en gardant le pouvoir de *scalabilité*, le client a à sa charge la mise à jour de l'ensemble des replicats. Nous allons aborder, succinctement, le déroulement d'échanges entre les clients et les serveurs pour voir comment cette mise à jour est réalisée.

Comme dans AFS, une base de données, permettant de retrouver la situation physique d'un volume, est dupliquée sur chaque serveur (un volume peut grossièrement être comparé à une partie d'une partition). Un volume est localisable grâce à son *VSG*[16] et est atteignable par son *AVSG*[17]. Lors d'un `open`, *Venus* (cf. AFS) choisit parmi l'*AVSG* de la ressource un serveur «maître» (le choix est soit aléatoire, soit analysé selon des critères de performances) afin de récupérer la totalité de la ressource. Cependant, un message est envoyé, au même moment, à chaque serveur contenu dans l'*AVSG* pour vérifier que le serveur maître contient bien la dernière copie. Lors d'une fermeture de fichier, ouvert en écriture, le client transmet les informations modifiées (toujours dans leur globalité) à tout l'*AVSG*. Afin d'être tolérant aux pannes, le client reconstruit périodiquement chaque *AVSG* correspondant aux ressources qu'il «cache». Dans ce but, il émet des requêtes de type «êtes vous vivant ?» au *VSG*. Si l'*AVSG* est augmenté (au moins un nouveau serveur est atteignable) suite à cette opération, l'ensemble des *callbacks* disposé par le client est retiré ; de cette manière, le client est sûr de récupérer la dernière copie des ressources lors d'un prochain accès. De même, lorsque l'AVSG est diminué, le client vérifie si parmi les serveurs qui ont disparu se trouve le serveur maître ; si tel est le cas, les *callbacks* sont évidemment supprimés (dans le cas contraire, aucune modification n'intervient).

Lors de la reconnexion d'une machine client (après une utilisation en mode déconnecté), un processus de resynchronisation des données est lancé par *Venus*. Celui-ci exécute une séquence d'opérations permettant de mettre à jour les différents serveurs. En cas de conflit (numéro de version dépassé, ressource effacée ...), un répertoire temporaire (du type `lost+found` sous UNIX) est créé pour y stocker les informations. Comme nous l'avons déjà précisé auparavant, un administrateur pourra, alors, réguler l'incohérence par la suite.

Les avantages de CODA, mis à par le haut degré de disponibilité, est la simplicité du serveur. En effet, en cas d'erreur, le serveur n'a rien à faire, le client se chargent de le mettre à jour lors de son retour à la normale. CODA est un DFS *scalable* et intéressant pour les utilisateurs qui sont prêts à accepter sa sémantique particulière (possibilité d'obtenir certains conflits nécessitant une intervention manuelle). Certains concepts intégrés dans CODA, font suites au DFS LOCUS (réplication optimiste ...) [LS90]. Par ailleurs, le client CODA est intégré au noyau Linux standards depuis plusieurs années.

Intermezzo

Le projet Intermezzo[18] s'inspire profondément du DFS CODA et est encore développé au CMU. C'est un projet *Open Source* dont le support client a fait son apparition

[16] *Volume Storage Group*, l'ensemble des serveurs ayant une image d'un volume.
[17] *Accesible Volume Storage Group*, les serveurs accessibles parmi le *VGS*.
[18] `http://www.inter-mezzo.org/`

dans le noyau de Linux depuis la version 2.4.15 (le noyau actuel est le 2.4.19). Affichant les mêmes objectifs que CODA, la conception a cependant été revue [BN99]. En effet, il est apparu que les échanges entre le noyau et le processus de gestion de cache (*Venus*) sont une perte considérable de performance. Le module noyau qui permet d'intercepter les appels systèmes en direction d'une ressource CODA, n'est pas optimisé. Lors de l'accès en lecture sur une ressource déjà présente dans le cache, le noyau transfère la requête au processus utilisateur *Venus* qui, à son tour, va rediriger la demande vers le noyau puisque la ressource n'a pas à être récupérée.

Inter-Mezzo a été conçu dans le but de supprimer ce surcoût inutile et pour cela un module noyau (appelé *Presto*) capable de filtrer tous les appels possibles et de les distribuer soit directement au système de fichier local soit au processus de gestion de cache (*Lento*, équivalent au *Venus* de CODA) a été créé. La figure 2.9 présente cette nouvelle architecture au sein du client.

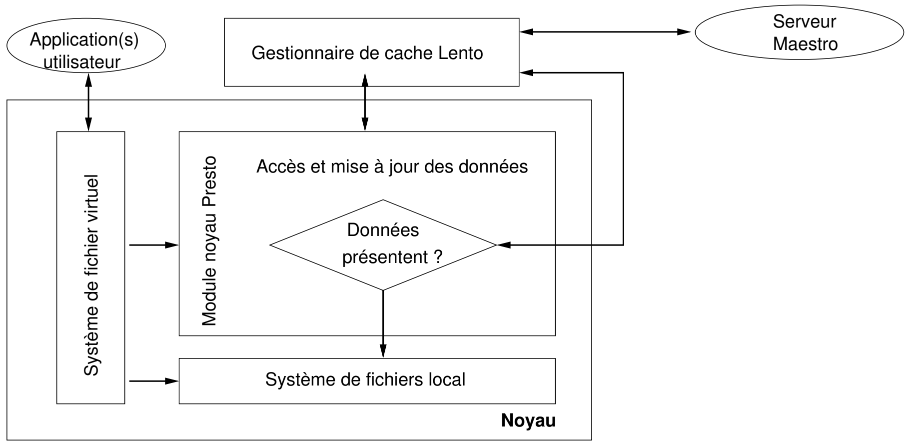

FIG. 2.9 – Architecture du client InterMezzo

Les *callbacks* sont implémentés aux deux niveaux (*Presto* et *Lento*). Lors d'une opération de lecture, le module noyau *Presto* vérifie s'il dispose d'un *callback noyau* sur la ressource, si tel est le cas la requête est dirigée vers le système de fichier local. Dans le cas contraire, *Presto* contacte *Lento*, via un appel synchrone, pour récupérer la ressource. En ce qui concerne toutes les opérations qui entraînent des modifications au sein des fichiers, *Presto* demande une permission d'écriture à *Lento* qui à son tour la demande au serveur. *Presto* alloue, alors, un tampon qui va contenir la journalisation de l'ensemble des changements. Une fois le tampon plein, *Presto* va le transmettre, de manière asynchrone, à *Lento* qui le fera parvenir au serveur. Les performances obtenues, grâce à cette méthode, sont proches de celle d'un système de fichier local (une fois que le client est en régime établi).

Le projet est très actif et une équipe de développeurs y apporte de nombreuses modifications toujours dans le but d'en optimiser les performances.

2.3 Modèles à accès direct

Les baies de disques actuelles (*SCSI*, fibre optique) utilisent des protocoles permettant à plusieurs machines d'accéder en concurrence aux diverses données stockées. Les systèmes de fichiers qui s'appuient sur ces architectures matérielles sont caractérisés par un modèle à accès direct[19] (chaque nœud lit et écrit de front sur le ou les périphériques de stockage, cf paragraphe 2.1). Ce type d'architecture, comme nous allons le voir, permet d'éliminer le phénomène de goulot d'étranglement, principal barrière au passage à l'échelle dans des systèmes à messages s'appuyant sur une gestion centralisée.

Par ailleurs, cette solution offre un haut niveau de disponibilité ; une défaillance sur un client n'empêche en aucun cas les autres machines d'accéder aux ressources.

2.3.1 GFS, *Global File System*

Le système de fichiers GFS [RRT96] a commencé à être développé en 1995. Dans un premier temps, il a eu pour but d'utiliser les capacités de la fibre optique afin de fournir aux applications scientifiques une solution efficace pour le traitement de leurs données à court et long terme. Aujourd'hui, il est principalement tourné vers les réseaux de stockages[20] dédiés. Un *SAN* est un réseau (SCSI ou de fibre optique, le plus souvent) interconnectant un ou plusieurs nœuds à l'ensemble des unités de stockages. Les protocoles utilisés pour le dialogue entre ces nœuds et ces phériphériques sont souvent spécifiques et nécessitent l'utilisation de matériels (disque, switch, câble) assez onéreux même si les performances obtenues sont excellentes.

D'un point de vue matériel, GFS permet d'abstraire en une seule unité (appelé *Network Storage Pool*) un ensemble de disques. Configurée au démarrage, cette structure est entièrement transparente d'un point de vue utilisateur ce qui permet de construire physiquement une architecture complexe (utilisation de différentes techniques *RAID*, regroupement en plusieurs «sous baies», ...) offrant un premier niveau de tolérance aux pannes.

La structure logique du système de fichiers diffère de celles vues auparavant (systèmes de fichiers usuels ou à la LFS) : l'ensemble des méta-données est réparti sur la totalité des unités de stockage. Le concept de *Resources Groups* (voir figure 2.10) est utilisé : les méta-données et les données sont réparties en sous groupes éparpillées sur l'ensemble des supports de stockage. Cette technique permet d'optimiser les accès parallèles par les clients ; un client peut lire des informations sur un disque pendant qu'un autre effectue un écriture sur une autre périphérique (nous rappellons que l'unité logique que visualisent les clients et physiquement composée de plusieurs unités de stockage).

Par ailleurs, afin d'améliorer l'utilisation de l'espace disque disponible au sein d'un bloc (*File System Block*) : le modèle de *Stuffed Dinode* a été implémenté (il s'agit d'occuper la totalité d'un bloc, réservé pour stocker des méta-données). Prenons l'exemple d'un bloc de 4Ko, les 128 premiers octets sont utilisés pour stocker les méta-données et les 3968 restant, habituellement stockés comme pointeur de redirection vers les données, contiennent les données des fichiers d'une taille inférieure à 4Ko. Après quoi, l'inode est utilisé comme usuellement. Cette méthode permet de réduire les accès

[19] En anglais : *Shared-disk File System.*

[20] *SAN, Storage Area Network.*

disques (en récupérant un bloc, il est possible d'obtenir la totalité d'un élément : répertoire ou fichier) et donc d'obtenir de meilleures performances. La structure de l'inode se différencie, de plus, en un autre point : là où des systèmes de fichiers utilisent une structure fixe de pointeur d'indirection [Bac86], GFS est conçu de manière à ce que l'ensemble des pointeurs, contenu vers l'inode et en direction des données, soit de même niveaux. Ce niveau évolue selon la taille du fichier (l'inode contient uniquement des pointeurs de simples indirections puis de double,...).

Une gestion avancée des espaces libres accélère les recherches et permet de positionner les métas et les données intelligemment. Une correspondance entre l'état de chaque bloc (libre ou occupé) est stockée au sein d'une liste [PBB+99] spécifique à chaque *Ressource Groups* (figure 2.10).

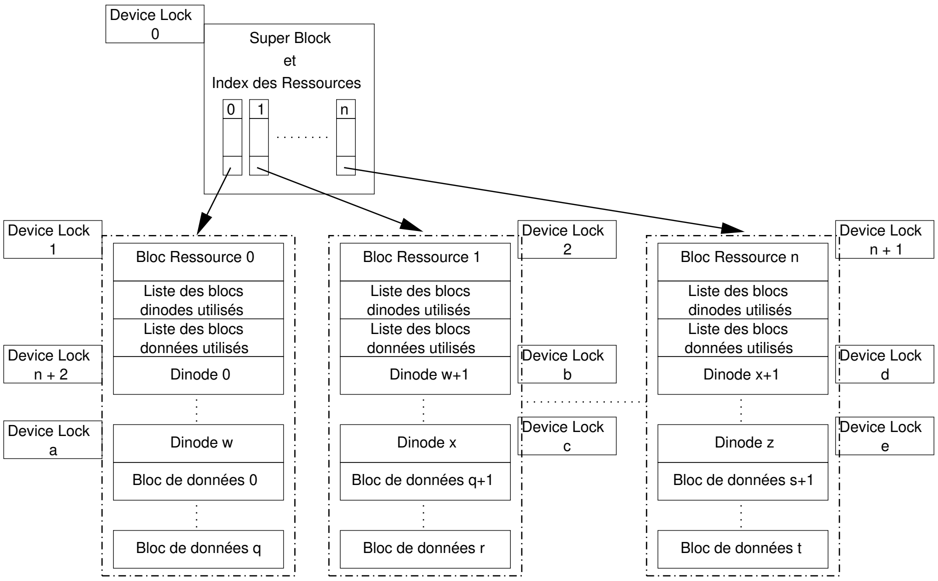

FIG. 2.10 – Structure logique utilisée par GFS

Les Dlocks (*device locks*) sont des mécanismes d'exclusion mutuelle permettant de maintenir la cohérence entre les différents clients et les données partagées. Implémenté au sein des disques, les *Dlocks* sont utilisés par des appels de commandes SCSI simples [RRT96] puis plus évoluées par la suite [PBB+00].

Un verrou (*lock*) est caractérisé par :

son état : pris ou libre,

sa portée : il est placé sur un unique, une partie, ou encore plusieurs blocs ;

son numéro de version : qui permet d'effectuer une gestion de cache ;

sa durée : un *lock* est pris pour une durée définie, ceci évite les interblocages (*deadlocks*) ;

un compteur : il permet de savoir combien de fois le verrou a été demandé par un même nœud.

La cohérence implémentée dans GFS est réalisée par l'accès partagé aux méta-informations. En effet, les verrous portent uniquement sur les méta-données (Zebra s'appuie sur ce concept également). Dans un premier temps, chaque *Dlock* correspondait à une demande en provenance d'un client ; si le verrou s'avèrait libre, il était alors attribué au nœud. Cela permettait d'obtenir uniquement des actions atomiques sur les méta-données. Par la suite, l'utilisation d'un numéro de version (incrémentée à chaque modification) a permis de mettre en place une gestion de cache en lecture sur les différentes machines composant le *cluster* (des requêtes en provenance des clients testent périodiquement cette valeur).

Afin d'éviter les problèmes d'interblocage, la notion de *back-offs and retries* a été incorporée au système : lorsqu'un processus souhaite accéder à plusieurs *locks* au même moment, il en acquiert un premier et si, au bout d'un certain temps, il voit qu'il lui est impossible de récupérer le suivant, il libère le premier *lock* qui doit lui même empêcher un autre nœud de rendre le second *lock*. Le nœud retentera de récupérer les verrous par la suite. La notion de temps apparaît de nombreuses fois au sein du système de cohérence de GFS. Un «chien de garde» (*time out*) permet de supprimer les locks bloquants, comme nous venons de le voir, mais aussi d'optimiser le cache. Lorsqu'un nœud souhaite garder un verrou plus longuement, il suffit qu'il émette périodiquement un message indiquant qu'il possède toujours le verrou (*Touch Lock*) ; de cette manière, le système est sûr que le client n'est pas défaillant ou arrêté.

Afin de limiter le nombre de demandes intempestives sur un verrou, un identifiant client a été ajouté. Lorsqu'une demande n'a pu aboutir, le DFS retourne, au nœud *demandeur*, le numéro du client qui possède actuellement le verrou. De cette façon les deux entités («demandeur» et «teneur») peuvent dialoguer directement. Toujours dans ce même objectif (éliminer les trames inutiles en direction du *Network Storage Pool*), une couche faisant correspondre un compteur par *dlock* a été ajoutée entre le système de fichiers et l'unité logique de disques. Cette couche, appelée *recursive dlock*, permet de savoir si un verrou est déjà tenu par un nœud et ainsi éviter de le redemander au niveau des disques. Enfin, un système de queue (*Conversion locks*), a été mis en place : il règle tous les problèmes de famine, puisqu'un verrou sera délivré au nœud dans l'ordre chronologique des demandes.

Une périphérique implémentant un *Device lock* pourra délivrer des *Dlocks* seulement après l'envoi d'un message spécifiant à l'ensemble du *cluster*, la mise à zéro de tous les verrous qu'il a sous sa responsabilité. Ceci permet dans le cas d'un dysfonctionnement survenant sur ce périphérique, de rester dans un état cohérent, puisque chaque nœud devra réitérer sa demande. Une solution beaucoup plus lourde, et donc inenvisageable, aurait été de stocker, sur les unités, l'état de chaque verrou.

Dans le cadre de la conception d'un système de fichiers distribué, nous avons vu qu'il est intéressant de prendre en compte les contraintes liées à la latence des disques (LFS a d'ailleurs proposé une nouvelle vision sur la disposition des informations). GFS, en plus d'avoir une structure de représentation qui diffère légèrement de la structure commune des systèmes de fichiers (*Ressource groupe*, *Stuffed inode*), utilise une technique (connue sous le nom d'*Extendible Hashing* [FNPS79]) permettant d'optimiser les recherches au sein d'un répertoire. En effet, lors d'une gestion simple (pointeurs d'indirections), le nombre d'entrées au sein d'un répertoire est tout d'abord limité et entraîne un temps non négligeable lors d'éventuels *lookups* (cf. Annexe B).

Comme nous avons pu voir, GFS utilise une gestion de cache ; celle-ci a pour avantage la minimisation des espaces disques et permet d'obtenir les taux de performances requis. Cependant, le principal inconvénient en découlant, est la complexité de la gestion de la tolérance. Toutes les modifications effectuées et contenues uniquement au sein du cache d'un nœud sont susceptibles d'être perdues lors d'un éventuel arrêt de la machine. Afin d'éviter la totalité des problèmes liés au cache et de ne pas utiliser de solutions trop lentes et trop gourmandes pour retrouver un état cohérent après erreur (vérification de la totalité des disques partagés), un module de journalisation a été implémenté au sein de GFS. Basé sur deux sous systèmes : le *Transaction* et le *Log Manager*, il permet de rendre atomique certaines opérations (*Transaction*) et d'être en permanence sur un état cohérent (*Log Manager*). La différence avec les systèmes de journalisation locaux et que chaque entrée du service de transaction doit pouvoir être retirée de façon totalement désordonnée puisqu'il est impératif de prendre en compte le fait que les données soient partagées entre plusieurs clients (*callbacks*). Dans le cas d'une éventuelle panne survenant sur un nœud, une des autres machines du *cluster* va récupérer puis exécuter le *log* créé par le client défaillant (nous rappelons qu'un «chien de garde» permet de détecter la perte d'une machine) qui se trouve sur l'unité logique partagée. Le système de fichier, retrouvera ainsi un état cohérent.

En 1998, GFS a été porté sous un environnement Linux (initialement sous IRIX une variante du System V d'UNIX), afin de profiter de l'accessibilité aux fonctions du noyau et de se tourner vers la conception d'un système de fichiers distribué plus général (un démon permettant d'abstraire les protocoles spécifques afin d'utiliser GFS sur toutes sortes de disques banalisés a d'ailleurs été intégré à la dernière version [PBB+00]). L'équipe de GFS travaille, actuellement, sur des outils permettant de gérer dynamiquement la prise en charge de nouveaux disques au sein du *Network Storage Pool*. Depuis août 2001, Sistina Software a décidé de changer la licence de leur produit GFS (sous licence libre auparavant), ce qui explique la création peu de temps après du projet OpenGFS[21].

D'un point de vue logiciel, GFS a introduit deux caractéristiques principales qui permettent d'optimiser les accès disques : le concept de la *Stuffed Inode* qui permet de minimiser le nombre de blocs du système de fichiers nécessaire pour le stockage d'une ressource et l'utilisation de l'*Extandible Hashing* pour accélérer la recherche au sein d'un répertoire. La gestion de la cohérence implémentée directement sur les disques est aussi une nouvelle approche, toutefois elle est spécifique au protocole et aux commandes SCSI et ne peut, par conséquence, pas être considérée comme une solution *scalable* aujourd'hui.

2.3.2 GPFS, *General Parallel File System*

GPFS [Sch02] est un système de fichiers d'IBM, développé depuis 1998, il a été initialement conçu pour répondre à un fort taux de débit sortant (*Read Only file System*) utilisé dans les serveurs vidéos [Sch96]. Comme GFS, il s'est tourné peu à peu vers des axes plus généraux dans le but de s'affirmer comme une solution potentielle des systèmes de fichiers distribués pouvant passer à l'échelle. Comme GFS, GPFS permet de visualiser l'ensemble des disques partagés, comme une unique et même entité. La structure physique peut s'appuyer soit sur un *SAN* de fibre optique ou de SCSI permettant de relier directement les disques aux noeuds

Une première différence avec GFS et la possibilité de modifier dynamiquement la structure physique de l'unité logique. Cet avantage considérable (rappelons que la

[21] `http://opengfs.org/`

structure du *Network Storage Pool* de GFS est configurée de manière statique lors du démarrage) permet de retirer ou d'ajouter des périphériques de stockages selon les besoins. De plus, GPFS inclut un mécanisme de réplication des données automatique, cette distinction supplémentaire propose, en plus de la tolérance aux pannes fournie par la baie de disque, un niveau de fiabilité supérieur.

Une des principales particularités de la structure logique du DFS GPFS est la taille inhabituelle des blocs du système de fichiers (en général de 256Ko mais pouvant varier de 16k à 1Mo). Ce choix découle du fait que GPFS a été conçu pour gérer de larges fichiers. L'accès par blocs plus grands réduit le nombre d'appel sur les E/S. Dans une seconde étape, et afin d'obtenir des performances sur l'accès à des fichiers de moindre taille, les blocs du système de fichiers ont été divisés en 32 sous-blocs.

Un module spécifique, appelé *Allocation Maps* est utilisé pour gérer les espaces vides au sein du système de fichiers. Toujours dans l'optique d'optimiser les accès parallèles, les informations représentant l'occupation des blocs sont réparties sur plusieurs régions. Ceci permet à plusieurs noeuds d'allouer des espaces en concurrence (puisque chacun peut accéder à une région particulière). Au sein du *cluster*, un nœud a la charge de maintenir les informations relatives à cette structure[22].

A la différence de GFS qui utilise une vue globale pour le partage des informations (chaque nœud comporte la couche GFS qui est elle même composée de l'ensemble des spécificités présentées), GPFS mélange deux approches pour améliorer les gains : la gestion de verrous distribués (GFS) et la gestion par point centralisé (la cohérence est maintenue par un nœud qui a la responsabilité de tous les accès aux informations). Ce compromis entre ces deux approches permet de réduire le surcoût de gestion en cas de conflit. Il a été observé qu'une approche centralisée peut parfois se révéler plus performante qu'une architecture totalement distribuée.

Le concept de granularité des verrous apparaît au sein de GPFS. Des verrous à portée plus petite vont entraîner un plus grand nombre de requêtes de *locks* ; en contre partie des verrous à portée plus large vont engendrer un plus grand nombre de conflits. De plus, GPFS propose un partage total sur les métas ainsi que sur les données. Ces deux types d'informations n'ont pas les mêmes caractéristiques (allocation de bloc, déplacement des données, ...) et ne peuvent être traités de manière identique.

Pour répondre à toutes ces contraintes, GPFS utilise trois sous modules :

***Byte Range Locking* :** Ce module fournit un accès partagé par plage. Prenons cet exemple, un premier nœud accède à un fichier, le *BR locking* lui remet un verrou sur la totalité du fichier (qui peut être distribué sur un bloc ou plusieurs et de même sur une unité ou plusieurs). Un seconde machine souhaite travailler sur le même fichier, il demande, alors au nœud qui a la charge des *BR locking* (*Token Manager Node*) de lui fournir le verrou sur ce fichier. Celui-ci étant dans l'impossibilité de lui remettre, lui retourne les identifiants des noeuds qui tiennent le *BR lock*, dans notre exemple l'unité 1. Le second nœud fait alors savoir à l'autre machine sa requête. Si le nœud 1 a fini avec le *token* (ou *lock*) il le passe à l'ordinateur 2. Dans le cas contraire, la machine 1 gardera un *lock* plus restreint correspondant à la partie sur laquelle il travaille, par exemple du début jusqu'au milieu du fichier et transmettra au second nœud un *lock* entre la moitié et la fin du fichier. La machine 2 avertira le *Token Manager* qu'elle vient de récupérer un *lock* sur telle

[22]Une autre particularité de ce DFS et qu'il utilise au sein du *cluster* des noeuds dédiés pour fournir certains services.

plage. Les noeuds dialoguent directement entre eux dans le but de «*soulager*» le *Token Manager*. Néanmoins, si certes il y a un bien un gain de performance à accéder à de larges blocs[23] pour un unique client ; le partage par plage, entre plusieurs machines, ne peut être qu'au minimum sur une distance d'un bloc[24]. Dans le cas contraire, il est impossible d'assurer la cohérence lorsque deux processus accèdent à la même information. Pour éviter ce problème, GPFS utilise une méthode permettant l'accès aux sous blocs : *Data Shipping mode*.

***Data shipping mode* :** cette méthode consiste à attribuer à chaque nœud du *cluster* un ensemble de blocs dont il sera responsable de la cohérence. Lorsque deux machines souhaiteront accéder à un même bloc, les demandes de lecture et d'écriture seront transférées au nœud responsable qui, le seul, sera apte à exécuter des opérations sur ce bloc. Cette technique permet de supprimer le surcoût lié à la finesse du grain. Une variante a été implémentée pour le gestion du partage des inodes, appelé *MetaNode*.

***MetaNode* :** GPFS propose un verrou en écriture partagé pour les inodes. Lorsqu'une machine récupère un verrou sur un inode (aucun autre nœud ne tient de *lock* sur ce dernier), le nœud devient la *metanode*. Comme pour le *data shipping*, elle devient responsable de la cohérence des informations et est la seule autorisée à lire ou écrire sur le disque. Lorsqu'une autre machine va vouloir accéder à la méta-donnée, le *Token Manager* lui renverra l'identifiant de la *metanode*. Celui-ci pourra alors récupérer une image de l'inode et mettra à jour périodiquement la *metanode* par l'envoi de messages. La libération de l'inode par la *metanode* entraîne une mise à jour sur disque. Les autres machines n'apercevant plus la *metanode* (retour de requêtes de mise à jour négatives) redemanderont l'inode au *Token Manager*.

GPFS intègre aussi la politique *Extendible Hashing* pour optimiser la gestion des «larges» répertoires. Toutefois, il inclut une légère variante qui consiste à stocker uniquement au sein de la table de *hash*, les valeurs qui pointent vers un sous ensemble. Ainsi, lorsqu'une nouvelle valeur est associée au sein de la table, un remaniement est effectué seulement sur le entrées concernées. Certains comparent la table de hash à un fichier à trou puisque les valeurs au sein de la table peuvent ne pas se suivre (par exemple : 00 puis 1001 puis ...). Le gain de place peut s'avérer rapidement considérable et donc la recherche plus rapide.

Comme GFS, GPFS incorpore un module de journalisation qui permet au système de fichiers de retrouver rapidement un état cohérent lors du dysfonctionnement d'un nœud. Dans le cas d'un partitionnement en deux dû à une panne du réseau au sein de la grappe, le groupe majoritaire de noeuds GPFS deviendra le nouveau système distribué. L'autre ensemble (minoritaire) se verra refuser l'accès à la baie de disques partagés, jusqu'à ce qu'ils puissent réintégrer le groupe majoritaire. Pour que le groupe majoritaire puisse utiliser un système cohérent, l'ensemble des *logs* des noeuds minoritaires sont exécutés.

Un critère prouvant la puissance de cet outil est qu'il est actuellement utilisé sur six des dix machines les plus puissantes au monde, avec comme porte flambeau l'ASCI White[25] du *Lawrence Livermore National Laboratory* avec ses 75 To d'informations.

[23] Rappelons qu'en moyenne les blocs du système de fichiers sont de 256Ko

[24] Il est intéressant de noter qu'il n'y a pas un *Dlock* par bloc au sein de GFS mais un nombre limité et réparti sur la totalité du *Network Storage Pool*

[25] Première machine au top 500 `http://www.top500.org/`

2.4 Protocole usuels et P2P

Plusieurs systèmes sont utilisés pour transférer des données à l'échelle du *web*, le plus connu étant sans doute le protocole HTTP. Dans les cadres des WANs, divers problèmes apparaissent : certains sont de nature purement technique (latence du réseau) et d'autres sont liés à l'activité humaine (la sécurité notament). Dans le cadre d'applications s'exécutant sur plusieurs sites s'ajoutent les contraintes de connectivité dûes à la présence de machines pare-feux bloquant tout ou partie des protocoles. C'est ainsi que par exemple, l'utilisation de l'UDP ou l'acceptation des connexions entrantes TCP sont la plupart du temps bannies (à part sur des machines dédiées comme les serveurs *webs*). La principale cause de ce phénomène et que la majeure partie des systèmes dits «pair à pair» actuels, [slddR01, Gnu, CSWH01] utilisent diverses techniques afin d'ouvrir (plus ou moins) involontairement des brèches dans les *firewalls*.

En plus des difficultés décrites ci dessus, ces modèles se tournent principalement vers la conception de méthodes performantes et *scalables* pour la recherche et la livraison rapide d'une ressource. Aucun n'intègre encore une notion de cohérence entre les images des fichiers et les utilisateurs et ne comportent pas, la plupart du temps, une arborescence structurée. Toutefois, ces systèmes peuvent être comparés comme des systèmes de fichiers *Read Only*, c'est pour cela qu'il est intéressant d'étudier les techniques et méthodes qu'ils intégrent (surtout au niveau de l'échange et de la gestion des méta-informations).

2.4.1 Le protocole HTTP

Ce protocole a commencé à être utilisé dès 1990. Il a été décrit dans le RFC1945 puis normalisée dans le RFC2616 [RFC97]. Ce protocole est un exemple de protocole client/serveur : un connexion TCP est d'abord initiée par le client afin d'avoir une requête, la réponse du serveur arrive alors en suivant le chemin inverse. La version actuelle permet, entre autres, de réutiliser la connexion ouverte afin d'envoyer plusieurs requêtes de manière séquentielle sans «repayer» le coût d'établissement d'une nouvelle connection. Un autre aspect intéressant est le support de l'extension *byte-range* (proche du concept introduit dans GPFS, cf. paragraphe 2.3.2) qui permet de récupérer uniquement une partie d'une ressource (pour terminer un téléchargement ou bien pour récupérer un fichier par morceaux sur plusieurs sites si celui-ci est dupliqué). Bien évidemment ce protocole est utilié principalement pour le téléchargement des pages web mais il est aussi utilisé par de nombreux systèmes d'échanges *P2P*[26] (KaZaA, Morpheus) afin d'en assurer les transferts de fichier.

2.4.2 Gnutella

Gnutella [Gnu] s'est proposé comme successeur de Napster lorsque les problèmes de *copyright* de celui-ci sont apparus. Il repose sur un algorithme de diffusion (*broadcast*) construit sur TCP/IP. Il n'y a pas de serveur au sein de l'architecture, toutes les machines ont un rôle symétrique. Une fois connecté au système Gnutella (un point d'entrée statique est supposé existé), il est possible de rechercher une ressource en émettant une simple requête à ses voisins ; ceux ci transmettrons de même la demande si il ne posède pas la ressource. Ce type de diffusion, appelé diffusion par innondations

[26]Litéralement *Peer To Peer*.

est très coûteuse, par conséquent l'utilisation d'un compteur de type *Time to live* est décrémenté à chaque fois que la requête passe d'un nœud à un autre. Lorsque cette indice atteint la valeur 0, la demande n'est plus propagé avec le risque de ne pas avoir trouver la ressource sur les différentes machines parcourues. Lorsque la ressource est trouvé les deux nœuds se connectent directement entre eux afin d'échanger les informations.

Une évolution du système a récemment ajouté l'utilisation de caches permettant d'optimiser les recherches : les nœuds plus puissant vont jouer le rôle de serveurs temporaires afin de soulager les machines plus lentes lors des opérations de recherches (les «serveurs» pourront temporairement garder un annuaire sur les ressources présentes sur d'autres machines). D'autres techniques que la diffusion par innondation (méthode ne permettant pas un passage à l'échelle correcte) ont été mise en œuvre (technique *Silverback,Tapestry*[slddR01]), toutefois aucune n'a été encore intégrée dans ce système. Le système Gnutella est amenée à disparaître si aucun changement n'est réalisé au sein de son implémentation.

2.4.3 Freenet

Le but de ce projet[27] est de fournir un service de partage de fichiers préservant l'anonymat des utilisateurs. Toujours fondé sur une architecture purement *P2P*, la charge engendrée par les différentes opérations est répartie entre les nœuds concernées. Chaque fichier est référencé par une clé unique : cette notion est très importante car elle pourrait permettre d'abstaire la structure et fournir un espace de nom global, comme dans AFS et ses descendants. Par ailleurs, une indépendance à la mobilité est proposée, les fichiers «migrent» entre les caches des diférents nœuds en fonction des demandes.

La clé qui permet d'identifier de manière quasi-unique un fichier est obtenue par une fonction de hachage appliquée sur une partie des données (la plupart du temps sur un descriptif succint du contenu du fichier). De cette façon, tout utilisateur peut accéder à l'information soit par une recherche directe avec la clé soit par un descriptif (un peu comme un moteur de recherche sur le *web*).

Une gestion de cache a été mise en place afin d'optimiser les performances. Ainsi, chaque client maintient une table dynamique lui indiquant les clés que possèdent ces proches voisins. Lorsqu'une opération de recherche est déclenchée, le nœud émet sa demande en direction du voisin qui contient la clé ou bien la plus proche de celle demandée. Cette procédure sera répétée comme dans Gnutella ; toutefois en plus du *Time to live*, la trame freenet contient un identifiant de requête ce qui permet d'éviter à un nœud de retraiter une demande et élimine ainsi tous circuits au sein du système (boucle réseau). La principale caractéristique qui différencie réellement freenet de Gnutella est la transparence de provenance de l'information. En effet, lorsqu'une demande aboutie, le nœud contenant la ressource n'est pas mis en relation directe avec le «demandeur» ; au contraire la réponse va transiter par l'ensemble des machines qui ont fait circuler la demande. De cette manière, chaque nœud va mettre à jour sa table (que nous pouvons comparer à une table de routage) et il est alors très difficile de savoir quelle machine a fourni la ressource. Dans le cas où la demande arrive sur une unité et ne peut plus être retransmise (*Time to live* à 0) ou qu'aucune clé ne correspond, la requête est renvoyée à l'émetteur qui la retransmettra à nœud ayant la seconde clé la plus proche au sein de sa table. Cette approche entraîne peu à peu une répartition de la structure en sous groupe

[27] `http://freenet.sourceforge.net/`

spécialisé dans certains documents (le fait d'émettre les demandes uniquement vers les machines ayant des clés proches et la propagation des clés engendrent ce phénomène).

Toutefois, ce système basé sur une gestion de cache ne permet pas de garder les informations indéfiniment. La taille bornée du tampon entraîne peu à peu la suppression d'information et des ressources peuvent donc totalement disparaître si aucun utilisateur n'y accède (gestion du cache en mode *LRU*[28]. De plus, le système de nommage n'est pas directement exploitable par l'utilisateur (système de clés peu convivial).

[28] Least Recently Used

Deuxième partie

NFSp

Chapitre 3

NFSp système de fichiers distribué

Le laboratoire ID-IMAG (Informatique Distribuée) est spécialisé dans la réalisation d'outils dédiés au calcul parallèle à haute performance et à leur validation sur des applications réelles. Une équipe du laboratoire s'est spécialisée dans l'étude et la mise en œuvre de techniques efficaces permettant le partage de ressources entre plusieurs applications parallèles au sein d'une grappe (mémoire partagée et système de fichier principalement). En d'autres termes, il s'agit d'exploiter au mieux les caractéristiques du réseau d'interconnexion (débit des liaisons, existence ou non d'arêtes disjointes ...) et des caractèristiques de chaque nœud (capacité de stockage, puissance CPU, ...). C'est dans le cadre de ces recherches que s'inscrit le projet NFSp.

Le système de fichier NFSp est né du besoin de disposer d'un outil performant et simple d'utilisation pour le partage des ressources au sein du *cluster* de 225 machines dont dispose le laboratoire depuis 2001. Comme nous avons pu voir dans la partie précédente, plusieurs gestionnaires plus ou moins performants sont disponibles ; néanmoins ils nécessitent souvent de remettre en cause l'architecture matérielle et logicielle de la grappe. Une autre solution, PVFS [CIBT], est développée depuis quelques années et vise à offrir un système de fichier distribué pour grappe se décomposant en un serveur de méta-données et des serveur de données. Ce modèle a des atouts remarquables, toutefois, l'installation d'un client impose le chargement d'un nouveau module noyau afin d'assurer une intégration dans le VFS Linux (*Virtual File System*, voir annexe A). Développé à partir du serveur NFS Linux en espace utilisateur, le système NFSp («boîte noire» composé de plusieurs sous modules) émule un serveur NFS et permet de partager, par conséquent, des ressources en utilisant les clients standards.

3.1 Présentation de NFSp

La vue d'ensemble, qui suit, est largement inspirée des articles [eYD02] et [LD02] rédigés par Y. Denneulin et P. Lombard, néammoins, une description même succincte du système est requise afin de bien visualiser et comprendre les divers aspects qui ont été modifiés et/ou améliorés (distribution puis réplication des serveurs de méta-données, mise en place de connexions fiabilisées ...).

3.1.1 Vue d'ensemble

Nous avons rappelé dans une première partie (cf. paragraphe 2.2.1), que le système NFS est performant uniquement au sein d'un groupe restreint de machines ; en effet, l'architecture clients/serveur centralisée, sur laquelle il s'appuie, est peu *scalable* sans investir d'importantes dépences. Le goulot d'étranglement imposé par le transit par le serveur de l'ensemble des données en est la cause. NFSp a été conçu dans l'optique de minimiser ce phénomène et d'utiliser, par ailleurs, la totalité (ou du moins le maximum) de l'espace disque disponible sur l'ensemble de l'architecture. Ainsi, le système (figure 3.1) se décompose en un gestionnaire de méta-données (ou méta-fichiers, démon nfspd), des serveurs d'E/S (ou de stockage, appelés *iods*) et des clients.

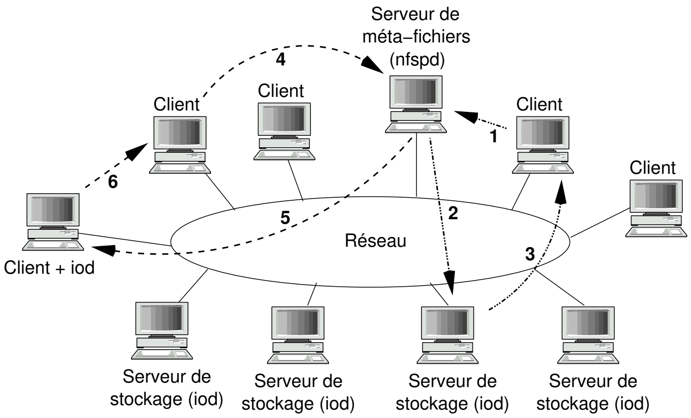

FIG. 3.1 – Architecture NFSp

Quand un client réalise une opération de lecture et/ou d'écriture, il émet une requête UDP-NFS à destination du gestionnaire de méta-fichiers (sur la figure 3.1, numéro 1 et 4). A la réception de la demande, le serveur transfère la requête en direction du serveur de stockage concerné (numéro 2 et 5) ; l'*iod* peut, alors, répondre directement au client (3 et 6) en se faisant passer pour le serveur (technique de *spoofing*). Cette dernière étape est importante puisque le client attend la réponse uniquement du serveur et non en provenance d'une autre adresse (nous rappellons qu'au moment de l'émission, le client a levé un *time-out*, de manière à réémettre sa requête dans le cas ou aucune réponse ne lui a été remise). La distribution des serveurs de stockage permet, ainsi, d'agréger les bandes passantes des *iods* durant les opérations de lecture (l'objectif étant d'atteindre des performances suffisantes pour saturer les cartes réseaux des nœuds «demandeurs») sans pour autant en modifier le client NFS.

Afin d'éliminer toute ambiguïté et de bien comprendre le rôle de chaque entité, nous avons besoin de définir les termes suivants :

Fichier de données : il contient les données «réelles» produites par les nœuds et est la ressource vu par chaque client. C'est ce fichier qui est réparti entre les divers serveurs de stockage.

Méta-fichier : les données de ce fichier sont invisibles pour l'ensemble des machines autres que le serveur de méta-fichier. Il contient, en plus des informations liées à l'inode (nom, propriétaire, estampille de temps ...), les renseignements requis pour pouvoir retrouver les données sur les serveurs de stockage, ainsi que la taille réelle du fichier de données.

Serveur de méta-fichiers : Ce service (*nfspd*), comme son nom l'indique, stocke la totalité des méta-fichiers au sein d'un répertoire local qu'il exporte par la suite grâce au protocole *mount*. Il a, par ailleurs, à charge la retransmission des requêtes aux serveurs de stockage.

Serveur de stockage : Il stocke les données transmises par le serveur *nfspd* et retourne les informations aux clients.

3.1.2 Implémentation

Cette partie présente, brièvement, certains points techniques liés à l'implémentation du modèle NFSp (figure 3.2).

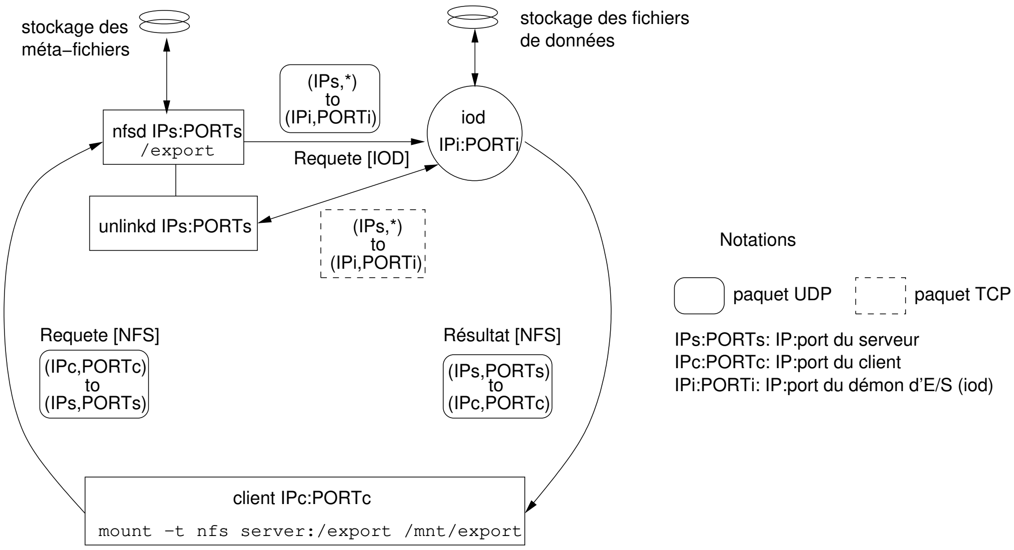

FIG. 3.2 – Fonctionnement du système NFSp

Lorsqu'un client créé un fichier, le méta-serveur lui fait correspondre un méta-fichier portant le même nom puis écrit au sein des données de ce méta-fichier les renseignements permettant de répartir le fichier de données. Pour le moment, seul un numéro de version et la taille réelle du fichier «client» sont stockés, néanmoins nous verrons par la suite, qu'il s'avère intéressant d'étendre ces informations pour mettre en place une meilleur politique de répartition (*striping*, équilibrage de charge, prise en compte des performances réseaux ...). Le fichier de données est ensuite envoyé vers les *iods*. Par la suite, lorsqu'une opération d'écriture sera effectuée, *nfspd* mettra à jour les données du méta-fichier (modification de la taille) pour transmettre par la suite les modifications «réelles» au(x) serveur(s) de stockage concerné(s).

Dans cette approche, une comparaison de l'arborescence des fichiers (par la commande «`ls -l`» d'UNIX, par exemple, à partir d'un client puis directement sur le gestionnaire de méta-informations) retournerait une structure identique (même nom, même hiérachie) ; toutefois, la taille différerait : les fichiers enregistrés sur le *nfspd* sont tous de même dimension (12 octets) alors que les clients aperçoivent la taille réelle.

Une première difficulté est apparue, lors de l'implémentation de l'opération de suppression d'un fichier. En effet, dans le cas du modèle NFS, le client émet sa demande d'effacement auprès du serveur qui élimine la ressource et retourne un accusé de réception (*ACK*) afin de valider l'opération (mode synchrone). NFSp répartit les données sur plusieurs fichiers éparpillés sur l'ensemble des *iods*, ce dernier ne peut pas, par conséquent, supprimer uniquement le méta-fichier correspondant (il est impératif de détruire les fichiers sur les serveurs de stockages afin d'éviter tous problèmes liés à une pénurie d'espace disponible). Un démon auxiliaire (*unlinkd*) a été implémenté afin de résoudre ce problème ; lorsque *nfspd* reçoit une requête de suppression, il retire le méta-fichier correspondant, retourne l'accusé au client et transmet une requête, via un simple *pipe*, au démon *unlinkd* qui se chargera de la suppression définitive des fichiers concernés sur les *iods* (effacement asynchrone).

Les *iods* sont des services «*multithreadés*». Un premier processus léger de haute priorité récupère tous les paquets UDP et les insère dans une file d'attente (*FIFO*) ; un second effectue la même opération pour les paquets TCP ; enfin, un dernier (ou plusieurs) analyse et exécute les requêtes en attente de traitement. Le protocole utilisé entre *nfspd* et les *iods* consiste en quelques messages en provenance du serveur de méta et en direction des serveurs de stockage :

- `READ` contient l'identifiant de la ressource à lire (inode, *offset*) et les informations permettant la reconstruction de la trame NFS renvoyée au client par la suite (identifiant de requête RPC, IP client ...).
- `WRITE` contient les mêmes données que la requête `READ` mais inclut les données à sauvegarder. Un acquittement est aussi émis vers le client.
- `PURGE` est utilisé pour supprimer les données invalides sur les serveurs de stockage (requête en provenance de *unlinkd* sous un flux TCP).

Deux autres messages sont utilisés pour l'administration et les dysfonctionnements.

Concernant la répartition des données sur les différents serveurs de stockage, une granularité de 64Ko a été choisie (copiée sur le modèle de PVFS). L'opération d'écriture au niveau du client NFS s'effectuant par bloc de 4Ko, les données sont transmises à un même *iod* durant 16 requêtes (pour les fichiers supérieurs à 64Ko). De même, une lecture NFS retourne 8Ko, au sein de l'implémentation ; par conséquent, un client souhaitant lire un fichier accèdera, indirectement, 8 fois à un *iod* avant d'«agresser» le serveur de stockage suivant. Comme nous avons pu voir durant toute la première partie, la granularité est un critère prépondérant dans les performances d'un DFS. Afin d'illustrer ces propos nous allons observer le cas le plus défavorable au sein de ce modèle (l'ensemble des clients souhaitent lire le même fichier au même moment).

Tous les client émettent des requêtes pour lire le debut du fichier sauvegardé sur un serveur de stockage (numéro 1 par exemple, sur la figure 3.3). Dès lors, après analyse par le gestionnaire de méta, tous les demandes lui sont transmises ; ce serveur se comporte alors comme un serveur centralisé NFS standard durant les 8 requêtes nécessaires pour chaque client (l'iod contient les 64K premiers octets et une requête NFS retourne 8Ko d'informations). Ainsi, aucune performance n'est gagnée durant cette étape (appelé phase de parallèlisation), un goulot d'étranglement apparaît sur l'*iod* en plus du surcoût généré par le serveur de méta-données. En supposant que les demandes soient validées selon un ordre séquentiel et qu'une seule soit traitée à la fois, le client 1 accédera à l'*iod* 2, une fois que ces 8 demandes auront été validées par le serveur de

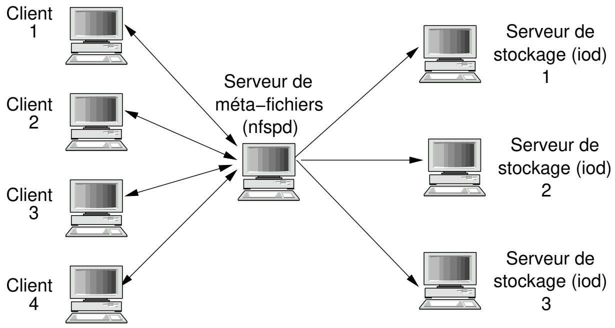

FIG. 3.3 – Parallèlisation des appels au sein de NFSp

stockage 1. Le client 2 pourra alors à son tour récupérer les informations sur l'*iod* 1 puis le client 3 et ainsi de suite. La granularité correspond à une certaine latence permettant au système de se mettre en place. A titre indicatif, nous avons choisi de faire apparaître la caractéristique des débits cumulés (figure 3.4) ; en effet, nous pensons qu'elle clarifie bien le concept.

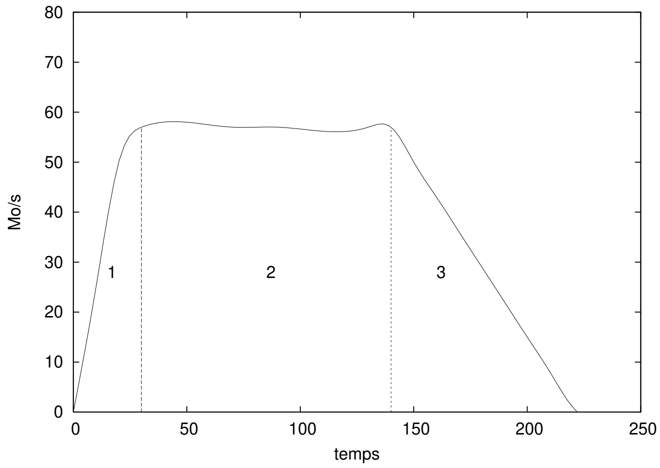

FIG. 3.4 – Caractéristique des débits cumulés dans NFSp

Phase 1 : le système se met en place, les clients se répartissent sur les *iods* ;
Phase 2 : régime stationnaire, toute les entités de l'architecture sont «stressées» ;
Phase 3 : certains clients ont finit de lire la ressource alors que certains accèdent seulement aux premières informations. Cette étape est beaucoup plus étalée dans le temps que la première ; ceci étant principalement dû à la politique du client NFS et du chien de garde (*time out*) qu'il utilise pour réémettre ses requêtes.

Il serait intéressant de continuer à explorer ce champ car le temps qui sépare l'arrêt des opérations de lecture entre le premier et le dernier client est tout de même considérable (sur une lecture par 32 client d'un fichier, le premier termine en un peu moins de 3 minutes alors que le dernier en plus de 9). Un test a été effectué avec une granularité de 8Ko pour la répartition des blocs, toujours est-il qu'aucune réelle différence n'a pu être observée. L'utilisation d'un fichier unique par serveur de stockage afin d'exécuter une seule ouverture de fichier et d'utiliser les performances du cache du système de fichier local a été envisagée et devrait être rapidement implémentée.

Nous venons d'aborder un aspect intéressant et difficilement «contrôlable» sans intervenir dans le code du client NFS. Néanmoins Les réels points «critiques» du modèle se révélent au niveau de la tolérance aux fautes. En effet, lorsqu'un serveur de stockage devient défaillant, les données qu'il stockait sont alors innaccessibles (disponibilité faible). Il en va également de même pour le serveur de méta-informations ; là encore, il est prépondérant dans la bonne marche de l'architecture, tout arrêt de celui-ci rend impossible l'accès aux ressources. De nombreux autres exemples sont disponibles, toutefois, le but de cette première version était de prouver l'intérêt de l'agrégation des débits dans une architecture de type grappe. Il sera donc important, par la suite, de regarder les différentes politiques pouvant être mises en place pour améliorer l'ensemble de ces aspects (bloc de parité à la Zebra, gestion journalisée des suppressions ...).

Enfin, nous insistons sur le fait que le goulot d'étranglement est d'autant plus présent lors des écritures. En effet, tout bloc de données en provenance d'un client transite obligatoirement par le serveur de méta-fichier. Il serait là aussi, intéressant d'étudier une éventuelle solution ; la distribution et/ou la réplication des méta-serveurs devrait fournir une première avancée en permettant de parallèliser certaines écritures.

Un des buts, lors de la conception de NFSp, était d'offrir la souplesse d'utilisation et d'administration qui est proposé dans NFS. Ce point a été largement atteint ; l'installation, décrite dans [LD02], s'avère simple (5 brèves étapes permettent de configurer la totalité du système).

3.1.3 Evaluation de la version NFSp existante

La version, présentée précédemment[1], correspond à l'avancement du projet NFSp lors de mon arrivée. La brève étude des performances [LD02] qui avait été réalisée, avant mon arrivée, avait permis de montrer l'efficacité du modèle dans le cadre de l'aggrégation des débits. Néanmoins, il s'avérait que le serveur de méta-données devenait très rapidement saturé (à partir de 16 clients et 16 *iods* ou jusqu'à 32 clients pour 8 *iods* sur les expériences effectuées) et se comportait, par conséquent, comme un goulot d'étranglement. Faute de temps, une étude plus poussée du modèle n'avait pu être réalisée ; toutefois, il est certain que le fait que le système ait été développé sur le code serveur NFS Linux en espace utilisateur[2] atténue les performances.

Afin de tester le système, nous avons choisi de nous appuyer sur le cas le plus défavorable, à savoir la lecture concurrente d'un même fichier d'une taille de 1Go (4 fois la taille de la RAM d'un nœud). Le fichier est créé par une simple commande `dd` et les données sont réparties sur 16 *iods* (cette opération prend un peu plus de 100 secondes en moyenne, soit un débit de entre 9 et 10Mo/s pour un maximun théorique de 12.5Mo/s). Les lectures sont lancées quasi-simultanément par le lanceur ka-run[3] sur les différents clients. Les résultats qui apparaisent sur la figure 3.5 ont tous été réalisés sur la grappe ID[4]. Ils diffèrent quelque peu des valeurs exposées dans [LD02]

[1] disponible à l'URL : `http://www-id.imag.fr/Laboratoire/Membres/Lombard_Pierre/nfsp/`

[2] *Universal NFS Daemon* - UNFSD

[3] `http://ka-tool.sf.net`

[4] `http://icluster.inrialpes.fr/`, cf. Introduction

où les tests avaient été réalisés avec une granularité de lecture de 4Ko. Toutefois, vu que l'ensemble des expérimentations ménées lors du DEA se sont appuyées sur une granularité de 8ko en lecture et 4Ko en écriture[5], le test a donc été renouvelé (courbe IP *spoofing*) dans l'intérêt de pouvoir comparer les résultats par la suite.

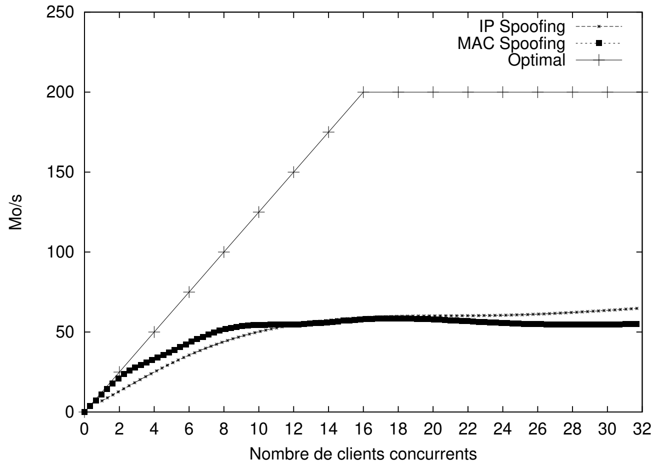

FIG. 3.5 – Agrégation des débits dans NFSp

A partir de 12 clients, la totalité des débits fournits par les serveurs de stockages arrête de croître et commence à stagner. Or, avec un débit de 12.5Mo/s théorique (chaque *iod* possède une carte ethernet 10/100 Mbit/s), les performances souhaitées avoisinnent les 200 Mo/s (16 *iods*, courbe Optimal). Ce comportement confirme la présence d'un goulot d'étranglement, ce qui est rapidement vérifié par une étude de l'utilisation CPU du méta-serveur (tableau 3.1). En effet, le gestionnaire de méta-données sature périodiquement à partir de 12 clients puis totalement à 16 et au dessus. Les ressources CPU sont réparties comme suit : 1/3 en espace utilisateur (NFSp) et 2/3 dans le noyau (gestion des E/S réseaux et disques). D'après l'étude précédente, menée par Pierre Lombard et Yves Denneulin, les E/S disques nécessité 10% de la CPU noyau sur *nfspd* et ils montrent par un bref calcul que la bande passante réseau ne peut être saturée. Il est vrai, après avoir analysé les débits échangés entre les entités de l'architecture, qu'aucune des cartes réseaux n'atteint un seuil critique. Toutefois, le nombre phénoménal d'interruption sur le serveur de méta-fichier laisse à penser que la pile IP du noyau Linux 2.4.4 ne peut supporter un tel travail. De plus, les multiples copies de données entre le noyau et l'espace utilisateur ont un coup important qui ne peut qu'accroître le phénomène. Aussi, il a été envisagé de faire passer très rapidement le démon *nfspd* en un module noyau chargeable (été 2002).

Une étude plus en profondeur de l'implémentation de NFSp durant ce DEA, a permis de montrer que la technique de *spoofing*, mise en œuvre (utilisation d'un socket `RAW` de niveau IP par le démon `iod` pour émettre la réponse vers le client tout en se faisant passer par le serveur), engendrait un surcoût imprévu. En effet, chaque paquet UDP *spoofé* était bien adressé à l'IP client, toutefois l'adresse MAC, affectée par le

[5]Nous insistons sur le fait que cette granularité est totalement indépendante du grain de répartition utilisé pour les données sur les *iods*.

Nbre clients	CPU Sys	CPU User	Nbre IN	Nbre CS
1	7%	3%	3000	2700
2	20%	10%	5500	4500
4	35%	17%	13800	3500
8	55%	25%	21000	500
12	60%	30%	21300	50
16	70%	30%	21500	40
24	70%	30%	21500	15

CPU Sys : CPU Système, CPU *User* : CPU utilisateur, IN : Interruption, CS : *Context Switch*
moyenne réalisée sur 1sec en régime stationnaire.

TAB. 3.1 – Ressource CPU du méta-serveur

noyau Linux correspondait à l'adresse physique du routeur. Par conséquent la totalité des flux transitait par la passerelle de la grappe qui est composé d'une carte ethernet de 1Gbits/s (soit 125 Mo/s), aucune expérience agrégant autant de débit n'ayant été effectuée, le problème était resté totalement transparent. Le système se comportait comme une «boîte noire» ayant un carte d'entrée à 100Mbits/s et une carte de sortie à 1Giga. Après correction (mise en place d'une socket RAW de niveau 2), un nouveau test a été réalisé (figure 3.5, courbe MAC *spoofing*). Là encore, la saturation du serveur influe sur les performances obtenues ; ainsi après 16 clients la technique du *spoofing* MAC est plus onéreuse que le fait de faire circuler les données par le routeur. L'échange supplémentaire entre le noyau et l'espace utilisateur, engendré par l'interrogation du cache ARP, s'ajoute à la charge CPU. Néanmoins, nous pouvons apercevoir que cette méthode offre de meilleurs gains lorsque le serveur n'est pas surchargé.

3.2 Composition de serveurs de méta-données

Comme il est indiqué, dans les paragraphes précédents, le modèle NFSp devrait pouvoir offrir des performances saturant quasiment le débit réseau disponible sur les nœuds. Malheureusement, l'approche centralisé du serveur de méta-données borne réellement le coefficient de passage à l'échelle du modèle. Il a d'ailleurs été observé, lors de la première étude, les limites de l'agrégation des débits et ceci était uniquement dû à l'utilisation totale des ressources CPU du méta serveur. Une distribution de la charge entre plusieurs serveurs est requise afin de prouver toute la puissance de l'approche NFSp. De plus, la distribution des méta-serveurs devrait permettre la parallélisation de certaines opérations d'écritures entre les divers serveurs.

La figure 3.6 illustre l'architecture souhaitée, il s'agit de regrouper en comité restreint des clients et un serveur de méta (AFS et ses descendants utilisent ce concept avec l'utilisation de sous clusters). Nous allons étudier deux méthodes permettant d'y parvenir, tout en respectant les contraintes du système NFS. Nous rappellons que les clients doivent absolument croire utiliser un serveur NFS, par conséquent nous sommes assujettis à fournir au moins, sinon mieux, l'ensemble des propriétés (sémantique, cohérence . . .) qui lui sont spécifiques. Par ailleurs, il est nécessaire d'apporter une solution qui complexifie le moins l'implémentation «simple» du système.

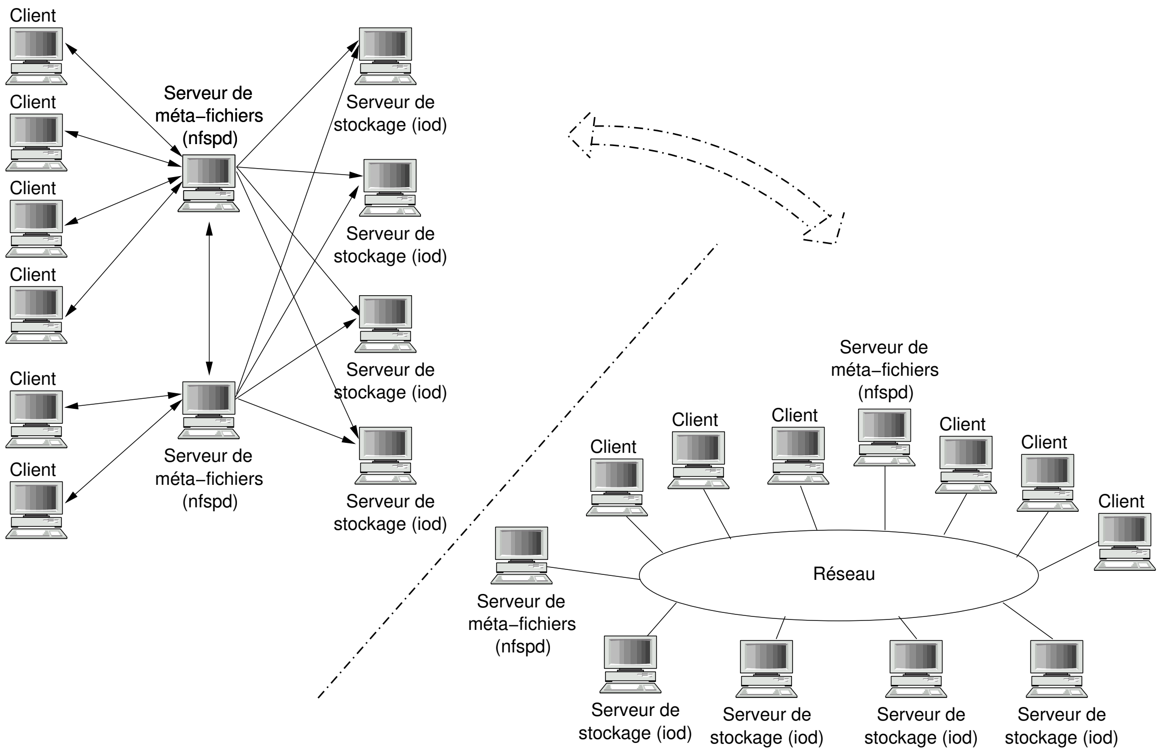

FIG. 3.6 – Composition des méta-serveurs dans NFSp

3.2.1 Distribution des méta-données

La première approche est fondée sur une répartition des méta-fichiers entre les différents gestionnaires. Un méta-serveur aura à charge les méta-données générées par un client de son groupe (technique du premier écrivain utilisé au sein de xFS, cf. paragraphe 2.2.2). Dans le cas, où un client souhaitera accèder à une ressource «distante» (dépendante d'un autre gestionnaire de méta-données) il s'adressera toujours à son «serveur». Ce dernier récupéra les renseignements nécessaires auprés du serveur distant afin de pouvoir restituer le fichier de données et transmettre les requêtes directement au(x) serveur(s) de stockage. Une variante aurait pu être de transmettre les requêtes du client au méta-serveur concerné (qui à son tour les aurait remis au(x) *iod(s)*). Néammoins, la première approche permet de «cacher» les informations (la sémantique NFS permet de «garder» les répertoires 30 secondes et les fichiers 3 secondes, cf paragraphe 2.2.1) sur le gestionnaire de méta «local» ; de cette manière les informations seront présentes lors d'un prochain accès intervenant dans cette période (diminution du trafic réseau).

Une autre solution fondée sur la répartition par identifiant généré à partir d'un CRC (méthode utilisé dans GFS et GPFS pour construire les tables de hachages, paragraphes 2.3.1 et 2.3.2) pourrait être employée afin de distribuer les méta-fichiers. Un ensemble borné d'identifiants possibles serait réparti selon le nombre de gestionnaires de méta-informations contenu dans l'architecture. Nous avons vu, lors de la première partie, qu'une répartition par fichier est souvent maladroite et entraîne des points de congestion selon le type de ressource (cf. Zebra méthode de *striping*, paragraphe 2.2.2). Malgré tout nous pensons que le stress généré par une tel configuration (tous les nœuds du système souhaitent accèder à la ressource `/etc/resolv.conf` par exemple) ne devrait pas saturer le serveur en charge de la ressource. La mise en place d'un «cache» sur les serveurs de méta-informations distants permettrait d'enregistrer temporairement les renseignements nécessaires pour la localisation des données réelles et par conséquent de limiter le nombre de requêtes.

Implémentation

Le module qui va permettre la distribution des méta-données se divise en deux sous entités :

Gestion de la transparence : La structure de l'arborescence des fichiers ne doit contenir aucune information concernant la situation physique des données et donc des méta-fichiers puisqu'ils reflettent la hiérarchie des fichiers de données. Selon la méthode employée pour la répartition des métas, la transparence offerte diffère. Ainsi pour la technique dite du «premier écrivain» nous avons une indépendance à la mobilité : chaque serveur devra interroger l'ensemble des méta-serveurs pour retrourver celui qui a en charge la ressource ; l'opération peut se révéler coûteuse mais permet de réguler dynamiquement la charge. L'approche par un calcul de CRC pour identifier un fichier offre une transparence de situation, certes moins souple lors de l'équilibrage de charge, mais beaucoup plus rapide lors de l'accès à une ressource (l'opération de *lookup* est réalisé localement). Une table permettant la correspondance entre un identifiant et sa situation serait repliquée sur chaque serveur ; *nfspd* n'aurait plus qu'à calculer le CRC pour retrouver quels serveurs interroger.

Partage des méta-fichiers entre les serveurs : un protocole doit être mis en place afin de permettre la partage des méta-fichiers entre l'ensemble des démons *nfspd*. Pour l'éxperimentation, nous avons tout simplement adopté le protocole NFS (figure 3.7). Partant du fait que de cette façon, la cohérence entre les méta-serveurs correspondra à la cohérence spécifique entre tous les clients. Chaque serveur exporte la partition où il stocke ses méta-fichiers et de même il «monte» les partitions des serveurs distants. La figure 3.8 représente une partie de l'arborescence d'un serveur de méta-fichiers : le répertoire `/export` est la partition exportée pour les clients tandis que `nfsp` contient les méta-fichiers classés selon les divers serveurs (nfspd 1 contient par exemple tous les métas fichiers du serveur local et les autres correspondent à des points de «montages» vers les serveurs distants). Le module de transparence permet d'abstraire l'ensemble de ces structures `nfspd X` pour la proposer dans le répertoire export accessible par les nœuds clients.

Expérimentation

Afin d'étudier les performances qu'apporterait cette approche, nous avons modifié le code pour tester le cas le plus défavorable. Ainsi, il n'a pas été utile de développer le module de transparence ; nous avons directement placé les différents points de «montage» dans le repertoire exporté. Chaque client accède indirectement à toutes les arborescences. Comme précédement, un client crée un fichier de 1Go (via la commande `dd` d'UNIX) dans le répertoire correspondant au méta-serveur du groupe au-

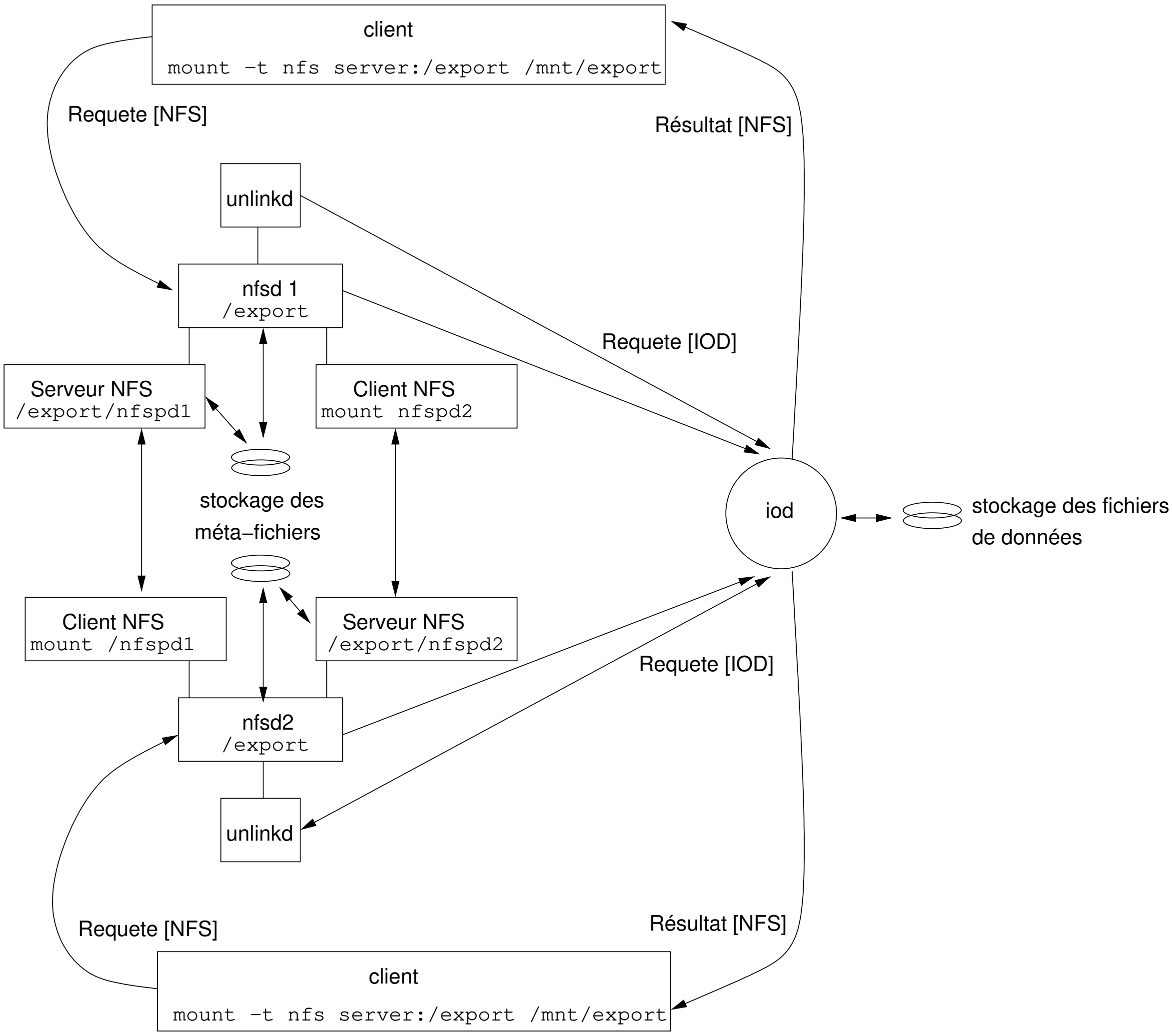

FIG. 3.7 – Composition des méta-serveurs dans NFSp via NFS

quel il appartient (`/export/nfsp1/FichierTest.txt`, par exemple). Une lecture concurrente est lancée sur l'ensemble des nœuds de chaque groupe. Les résultats qui apparaissent sur le graphique 3.9 s'avèrent être prometteurs ; en effet, la bande passante double entre les tests avec 1 méta et celui avec 2. Toutefois, nous apercevons, là encore, que le système atteint ces limites aux alentours de 20 clients ; la courbe arrête de croître et se met à stagner. Ce résultat ne nous a pas paru illogique tout d'abord. En effet, cette configuration atteignait ses limites (environ 45 Mo/s) autour de 12 client lors du précédent test (un unique méta). Avec 2 méta, l'architecture se comporte en deux sous clusters et par conséquent nous pouvions nous attendre à ce que les deux serveurs soient saturés vers ces valeurs. Nous avons donc effectué deux autres tests : avec 4 méta-serveurs puis avec un nombre égal de gestionnaires et de clients (chaque client accède à son propre méta). Malgré cela, il est apparu que le système atteint les mêmes bornes (environ 105 Mo/s). Par ailleurs, ces courbes (4 métas et *Equal méta(s)*)

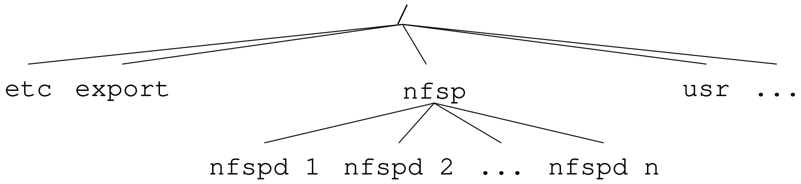

FIG. 3.8 – Exemple d'arborescence d'un méta serveur : les répertoires nfspd sont des points de montages sur les serveurs distants

nous permettre de remarquer le surcoût engendré par l'échange d'informations entre les démons *nfspd*. Ainsi, les débits obtenus s'avèrent être plus faibles lorsque le nombre de clients augmente (au dessus de 26). Ce qui n'est pas acceptable pour un passage à l'échelle performant. Dans tous les cas, les chiffres résultant de ces expériences sont toujours assez loin de l'optimal (même s'ils en sont proche pour un nombre de clients inférieur à 8). Le nombre d'interruption avec 2 serveurs de méta-fichiers reste assez élevé (les CPUs sont saturés au dessus de 24 clients, pour un nombre d'interruption de l'ordre de 20000 et des changements de contexte aux alentours de 400).

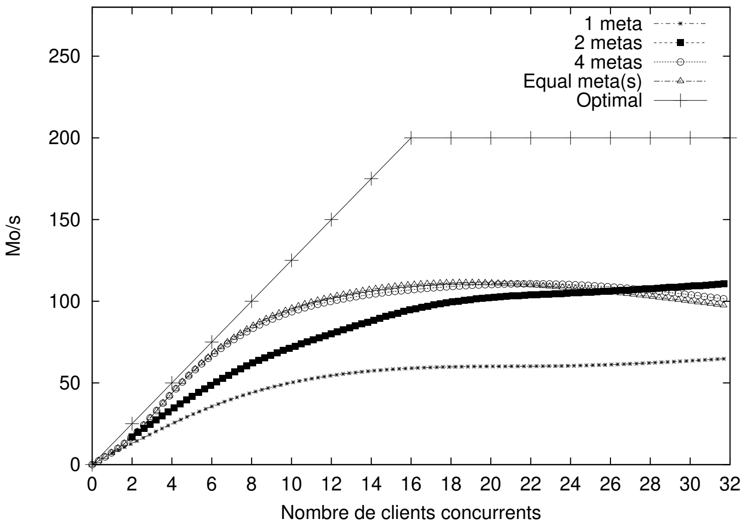

FIG. 3.9 – Agrégation des débits dans NFSp : distribution des métas serveurs

La quantité d'informations échangée pour la gestion du système est aussi importante : un client émet un peu plus de 130 000 requêtes NFS en direction du serveur (1 Go pour le fichier divise par 8 Ko). Sachant qu'une requête a une taille proche de 128 octets, le client envoie un peu moins de 17 Mo sur le réseau. A son tour, le serveur de méta-données transmet les demandes au sein de paquets d'une taille de 108 octets (soit un peu plus de 14 Mo pour lire le fichier de 1Go). Partant de ce point, pour qu'un client lise la totalité du fichier environ 31 Mo circule sur le réseau uniquement pour accéder à la ressource. En conclusion, le surcoût engendré par la lecture concurrente de 32 clients génére au minimum plus de 960 Mo de trames qui sont soit analysées soit construites par le serveur de méta ; un équilibrage correct serait de diviser ce nombre

Serveur	Debit In	Debit Out	Total
1	311	282	593
2	140	125	265
3	140	125	265
Total	591	532	1123

Les quantités sont exprimées en Mo.

TAB. 3.2 – Débit utile pour la lecture concurrente d'un fichier de 1Go par 24 client réparties sur 3 métas

entre les méta-serveurs. Une analyse de la quantité de données reçue et émise (tableau 3.2) a été réalisée sur la configuration suivante : 24 clients, 1O *iods*, 3 serveurs de méta et donc environ 750 Mo de requêtes à partager en 3.

Pour commencer, la quantité de données est largement supérieure à la valeur attendue (plus de 1 Go). De plus, la répartition des débits n'est en aucun cas proportionnelle, le serveur 1 qui a en charge le méta-fichier reçoit et engendre plus du double des données transmises par un des autres serveurs. Un *iod* reçoit environ 1.4 Mo dans ce cas présent (14 Mo pour un fichier de 1 Go divisé par 10 *iods*) pour un client. Sachant qu'il y a 8 clients par gestionnaire de méta et 10 serveurs de stockage, un démon *nfspd* devrait transmettre environ 112 Mo en débit sortant. Nous pouvons donc déduire que les serveurs 2 et 3 émettent aux alentours de 13 Mo chacun pour récupérer les méta-informations sur le serveur 1. Malheureusement, cela n'explique pas pourquoi, le premier serveur de méta-données reçoit et génére autant d'informations. Il serait intéressant d'étudier les limitations de la pile IP du noyau Linux 2.4.4 ; le trafic étant en mode non connecté plusieurs paquets pourraient être «droppés»[6] et les clients seraient donc amenés à les réenvoyer. Toutefois, par manque de temps, cette étude n'a pu être ménnée.

3.2.2 Passage à l'échelle

Un derniere experimentation a été conduite dans le but d'observer le comportement lors d'un passage à l'échelle plus important. Nous avons été limités par le support physique de la grappe : le cluster de 225 machines est construit sur un réseau 2Gbit (5 *switches*, connectés en anneau, interconnecte chacun 45 nœuds). De ce fait, un maximun de 20 serveurs de stockages est possible (20 *iods* à 12.5Mo fournit une bande passante cumulée à 250Mo soit 2Gbits[7]. Ce dernier graphe (figure 3.10) présente les résultats obtenues pour une lecture concurrente par 64 clients avec 20 *iods*, le nombre de méta-serveurs utilisés varie de 1 à 12.

Avec 12 méta-serveurs, le débit obtenu atteint plus de 75% du débit maximum (191 sur 250Mo/sec). Toutefois ce chiffre reste faible si nous nous plaçons du coté des clients : un nœuds reçoit 3Mo de données par seconde alors qu'il pourrait en traiter plus de 12. Néanmoins, les bornes physiques de l'architecture réseau sont atteintes et l'opération est réalisable à partir de 2 méta-serveurs (un seul méta-serveur retourne un *Time out* critique sur plusieurs clients et la valeur n'est donc pas exploitable). A titre indicatif, une lecture concurrente d'un fichier de 1Go par 32 clients sur le serveur NFS

[6]Lorsque le noyau est surchargé, il élimine les paquets qu'il ne peut pas analyser.

[7]Nous rappellons que les débits réseaux sont exprimés par puissance de 10 et non de 2.

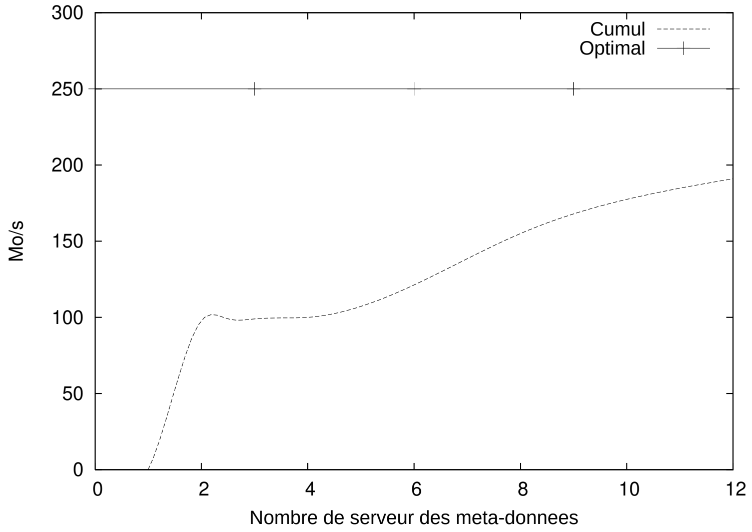

FIG. 3.10 – Agrégation des débits dans NFSp : distribution des méta-serveurs

standard de la grille[8] prend plus d'une heure ; cette même lecture par 64 clients a pris un peu moins de 6 minutes.

3.2.3 Réplication des méta-données

L'étude de la distribution des méta-données a révélé un surcoût important ; en effet, les requêtes «internes» (entre les méta-serveurs) ne sont pas assez «capitalisées» (trop de redondance). La lecture du fichier de méta-données nécessite de renouveller la demande toutes les 3 secondes et par conséquent engendre des «dépenses» trop importantes. Par conséquent, une seconde solution a été proposée.

Cette méthode consiste à repliquer le méta-fichier sur l'ensemble des méta-serveurs. En effet, ce fichier est d'une taille très petite (12 octets) et est rarement modifié (seul l'attribut spécifiant la taille du fichier peut être amené à évoluer pour l'instant). Partant du fait qu'une opération de lecture est renouvelée plus souvent qu'une écriture, il serait plus intéresssant de transmettre la requête de création à l'ensemble des méta-serveurs afin que chacun crée le méta-fichier correspondant (il en va de même des opérations de modifications altérant les méta-informations). De ce fait, nous proposons une indépendance à la mobilité puisque tout gestionnaire de méta-données possède la totalité de l'arborescence. De plus, les problèmes liés à l'implémentation d'un module permettant de rendre transparent les différentes structures sont éliminés : tous les serveurs exportent un répertoire qui contient la même arborescence (espace de nommage unique). Enfin, la mise en place d'un protocole permettant de transmettre les requêtes de création et de modification par un procédé de type *multicast*[9] en direction de tous les

[8] P3-1Ghz, 512 Mo de RAM et 1Go de swap sur disques SCSI utilisant xfs 2.4.5 comme système de fichier local.

[9] Emission d'un message à parti d'un nœud et en direction de plusieurs machines spécifiques ; différent du *broadcast* ou des méthodes de diffusion par innondation souvent utilisées dans les systèmes «pair à pair», cf. paragraphe 2.4.2.

gestionnaires est requis. L'utilisation du protocle NFS peut être adoptée dans un premier temps, toutefois, il apparaît que l'ensemble des informations contenues dans ses trames n'est pas forcément utile pour ce type d'échange (protocole spécifique, basé sur du RPC et contenant un grand nombre d'identifiants inutiles). Le passage à l'echelle d'un tel système (toujours avec une cohérence à la NFS) ne devrait pas poser de difficultés majeures : la mise à jour de 1000 méta-fichiers nécessite l'envoi de 12Ko ; en supposant que l'architecture est composée de 1000 méta-serveurs, cela imposerait un flux de 12Mo sur le réseau (ce qui est négligeable sur les interfaces réseaux actuels). Même si la solution n'a pu être testée faute de temps, elle apparaît comme celle qui va être intégrée au sein de l'architecture.

Enfin la mise en place d'un module, permettant au client le basculement rapide sur un autre méta-serveur dans le cas où le «sien» serait défaillant, apporterait un degré de tolérance aux fautes non négligeable. L'arborescence étant identique, l'utilisateur final, coté client, ne s'apercevrait pas du changement (utilisation d'une méthode de basculement proche de celle employée pour les serveurs *DNS*, qui ont un serveur primaire puis un secondaire ...).

Chapitre 4

Bilan

Nous avons présenté le système de fichiers NFSp développé au laboratoire ID, depuis plus d'un an. Ce modèle comportait plusieurs limites ne lui permettant pas d'atteindre ses objectifs, même si les résultats obtenus, en comparaison avec le modèle centralisé NFS standard, étaient largement plus élevés. Le principal problème était la centralisation des méta-données qui engendrait un goulot d'étranglement au niveau du serveur (cela se révélait par une saturation des ressources CPUs). La distribution des méta-informations a permis de réduire ce phénomène et a «offert» les débits proches des capacités optimales du réseau.

La mise en place de méta-serveurs entre deux (ou plusieurs) sites distants pourrait être envisageable (en utilisant un protocole de type «pair à pair» permettant de passer outre les pare-feux). De cette façon, le modèle pourrait s'étendre aux grilles. De plus et toujours dans le but de réaliser ce passage de manière performant, une étude plus approfondie aux niveaux des méthodes de réplication des données (migration, *pipeline*) pouvant être mise en œuvre serait intéressante à mener. Par ailleurs, nous avons vu qu'il s'avère souvent maladroit de répartir les données réelles sur une quantité trop importante de serveurs de stockages. L'utilisation de sous groupes d'*iods* incluant des blocs de parité (à la xFS) pourrait être une bonne solution.

Une fiabilisation des requêtes entre les serveurs de métas et les serveurs de stockage a été mise en place et est en cours d'expérimentation. Le but est de comparer les performances avec la solution UDP et de conclure sur la quantité de données réellement utile pour l'échange des informations (nous rappelons que le trafic réel est quasiment égal au double de la valeur théorique).

Enfin, lors du précédent chapitre, nous avons émis la possibilité de mettre en place une technique de *switch* entre les divers gestionnaires de métas. En conclusion, nous préciserons qu'il serait, là encore, avantageux de prendre en compte dynamiquement des nouvelles ressources (configuration à la volée) : un méta-fichier est représenté par 12 octets ; par conséquent, la réplication de la totalité de l'arborescence des fichiers ne sera pas une opération trop coûteuse (pour 1000 fichiers : 12 Ko). L'interaction de ces deux méthodes permettrait d'effectuer un équilibrage dynamique du système : si les serveurs deviennent surchargés, un nœud est alors choisi et devient à son tour gestionnaire de métas (cette opération ne prendrait dans notre exemple de 1000 fichiers, pas plus d'une seconde pour le transfert et 4 secondes pour la création de la structure sur le disque distant). L'auto-organisation de ce système de fichiers fera partie de nos objectifs par la suite.

Conclusion

Les systèmes de fichiers distribués permettent depuis plus de 20 ans de partager des ressources au sein d'un réseau local (ou mondial). Le principal intérêt de cette étude a été d'étudier et d'analyser différentes méthodes de placement et d'accès aux méta-informations au sein d'une architecture distribuée. Le placement a un influence forte sur les performances ainsi que sur le coefficient de passage à l'échelle. Nous avons d'ailleurs pu observer dans le cas du modèle NFSp que la distribution des méta-données permettait d'atteindre les bornes physiques en terme de débit du réseau sous-jacent.

Les systèmes de fichiers vont de plus en plus se scinder en deux catégories totalement différentes : d'un coté les solutions «centralisées» sur architectures dédiées (par exemple, *GPFS*) offrant des débits performants mais dépendants du matériel sous jacent ; et de l'autre les modèles «distribués» pouvant être installés sur tous types d'architectures matérielles. Néanmoins, il est quasiment certain que l'une et l'autre devront être capable de soutenir les débits engendrés par les applications futures. A titre d'indicatif, le CERN a annoncé qu'il aurait besoin, aux alentours de 2006, d'un système capable de partager des ressources, générées par le détecteur de particule, de l'ordre du Tera et même du Peta octets sur une architecture de type grille.

De nombreux projets travaillent déjà sur la mise en œuvre d'outils permettant de fournir des performances succeptibles d'assumer de telles quantités de trafic. Plusieurs systèmes, comme *GDMP* [SSA+01] par exemple, s'appuient de plus en plus sur le fait que les utilisateurs accèdent à une ressource uniquement en lecture une fois celle-ci créée. Ainsi, les points forts de ces modèles s'orientent autour de politiques de distribution et/ou de duplication des données optimales. Plusieurs nouveaux niveaux d'abstractions sont mis en œuvre ; parmi elles, la notion d'objet intègre les systèmes de fichiers : seule une partie des informations, en l'occurence «les objets», contenue dans un fichier est dupliquée sur le site distant puis réintégrée au sein d'une nouvelle ressource persistante. A l'opposé, des projets comme JetFile [GWP99] poursuivent les travaux introduits par CODA en construisant des solutions fondées sur l'approche optimiste de la réplication (utilisation de plusieurs replicats en mode écriture partagée) : ils partent du fait qu'un conflit de versions pour une même ressource (deux écrivains accèdent aux mêmes données en écriture) apparaît rarement et qui plus est, peut être corrigé automatiquement. Cette méthode permet d'offrir un système de fichier distribué proposant une réelle sémantique de partage qui peut se rapprocher, moyennant un certain coût, de la sémantique d'UNIX.

Par ailleurs, les systèmes d'échanges de fichiers dits «pair à pair» au niveau du *web* proposent des techniques permettant de retrouver, avec un moindre coût, des ressources. Des solutions comme *Costore* [CNY01] essaient d'intégrer certains concepts des modèles *P2P*[1] (utilisation de protocoles *multicast*) afin d'offrir des solutions pouvant s'appliquer sur un plus grand nombre de nœuds. Il serait intéressant d'approfondir l'ensemble de ces méthodes pour observer si elles peuvent réellement s'affirmer comme d'éventuelles solutions pour la distribution des méta-données et des données à l'échelle de la toile.

Toutefois, l'ensemble de ces systèmes de fichiers distribués ne prennent pas en compte la totalité des contraintes physiques de l'architecture sous-jacente. Le développement de solutions capables de s'adapter dynamiquement aux capacités disques, CPU ou encore aux arêtes du réseau est l'axe que nous avons choisi de continuer à étudier par la suite. Au moment où de nombreuses équipes travaillent sur des outils permettant l'équilibrage de charge CPU (*MOSIX*) ou encore le déploiement optimisé de processus, la création d'un système de fichiers s'appuyant sur un configuration «dynamique» qui permettrait d'améliorer considérablement le temps nécessaire à l'éxécution d'une application pourrait devenir un critère prépondérant au sein des architectures de type grappe ou grille. Son développement sera notre problématique avec pour objectif la mise en place d'une politique de régulation implicite permettant d'optimiser l'utilisation de chaque nœud. L'étude d'un système proposant ces différents services sans pour autant délaisser les caractéristiques liées au passage à l'échelle sera notre principal objectif.

[1]Litérralement, *Peer To Peer*

Annexe A

Virtual File System

Le rôle de la couche *VFS* est d'agréger tous les systèmes de fichiers présents au sein d'une machine. Lors de l'accès à une ressource, l'opération passe au travers de trois couches, la plus haute, appelé couche des appels systèmes gère les appels `OPEN`, `READ`, `WRITE`, `CLOSE` connus. Après avoir analysé l'appel et vérifié ses paramètres, elle appelle cette seconde couche qu'est le système de fichiers virtuel. Son rôle est la gestion d'une table analogue à la table des inodes pour les fichiers ouverts sous UNIX [Bac86]. Dans le système UNIX, un inode est repéré par un doublet (périphérique et identifiant) ; la couche *VFS* crée une entrée, appelée vnode (*virtual inode*), pour chaque fichier ouvert. Ces identifiants permettent de distinguer les fichiers locaux des fichiers distants. Dans ce dernier cas, ils fournissent les informations requises pour y accèder. L'opération `mount` habituellemnt utilisée dans le système de fichier NFS permet d'ajouter au VFS l'entrée correspondant au répertoire distant.

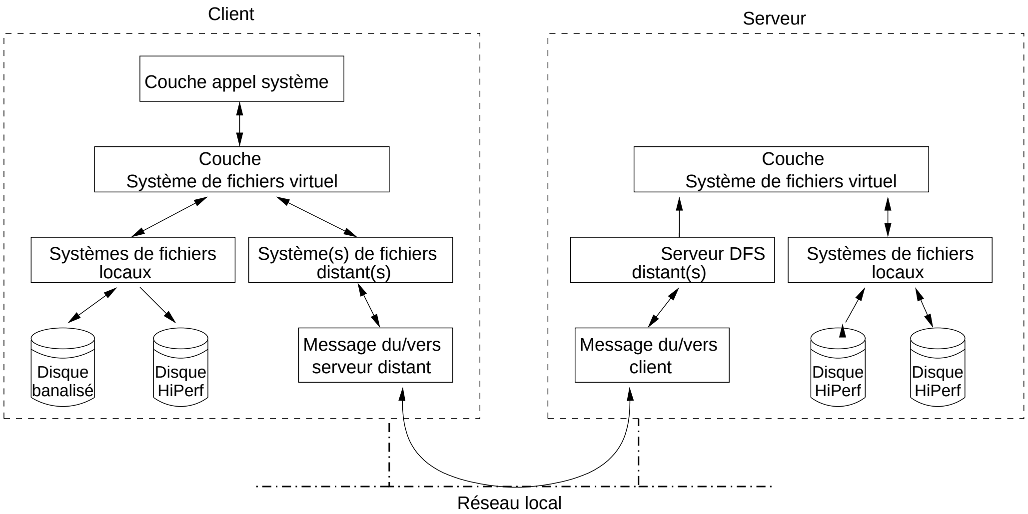

FIG. A.1 – Structure en couche des appels systèmes

Annexe B

Extendible Hashing

Extendible Hashing fournit une technique de stockage pour les données des répertoires permettant de retrouver rapidement et en un accès limité d'E/S, une entrée (un sous répertoire ou un fichier). La méthode est basée sur une fonction de hachage (CRC) permettant de répartir uniformément les différentes clés au sein d'une table (voir figure B.1). Chaque valeur de la table pointe vers un ensemble d'entrée de répertoires (*Leaf block and Direcory Entries*). Pour retrouver une entrée au sein du répertoire, il faut calculer la fonction de hachage qui génère une clé sur 32 bits à partir du nom du fichier. Ensuite en récupérant les n bits significatifs (poids faible), il est possible d'accéder directement au sous ensemble voulu (*leaf block*) et de regarder si l'entrée est présente ou non.

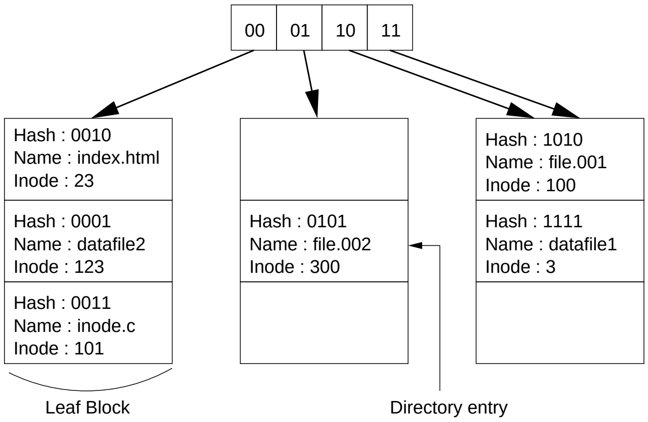

FIG. B.1 – Représentation d'une table de hachage et des pointeurs vers les blocs de données

La principale difficulté au sein de ce modèle est la possibilité d'une croissance rapide de la table de hash. En effet, une fois qu'un des sous ensembles est plein, il faut le scinder en deux groupes identifiables par une seule valeur au sein de la table de hash. Par exemple, sur la figure B.1, si un nouveau fichier est créé au sein du répertoire avec comme clé 0000, il faudra diviser le sous ensemble de gauche et créer une nouvelle entrée au sein de la table de hash qui doit toujours avoir une taille correspondant à une puissance de deux. Ainsi, pour une mauvaise fonction de hash, la table risquerait de grandir inutilement, puisque de nombreuses valeurs pointeront sur le même sous ensemble (sur la figure les deux pointeurs de droite). Néanmoins, cette technique permet de retrouver en une lecture sur un bloc de 4Ko plus de 1700 entrées on peut donc facilement imaginer les possibilités que propose un telle technique sur des blocs de plus grandes tailles ou encore en utilisant simplement des pointeurs de simples indirections pour stocker la table de *Hash*.

Bibliographie

[ADN+95] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli et Randolph Y. Wang. Serverless network file systems. *Département des Sciences de l'Informatique, Université de Californie à Berkeley, CA 94720*, 1995.

[Bac86] Maurice Bach. The design of the unix 0perating system. *Prentice-Hall*, 1986.

[BN99] Peter J. Braam et Philip A. Nelson. Removing bottlenecks in distributed filesystems : Coda and intermezzo as examples. *Université de Carnegie Mellon et Université Ouest de Washington*, 1999.

[CIBT] Philip H. Carns, Walter B. Ligon III, Robert B. Bross et Rajeev Thakur. PVFS : A parallel file system for linux clusters.

[CNY01] Yong Chen, Lionel M. Ni et Mingyao Yang. A large-scale reliable storage service based on peer-to-peer clusters. *Rapport technique MSU-CSE-01-31, Départment des Sciences de l'Informatique, Université de l'état du Michigan,* Décembre 2001.

[CSWH01] I. Clarke, O. Sandberg, B. Wiley et T. W. Hong. Freenet : A distributed anonymous information storage and retrieval system. *Science de l'Informatique, volume 2009*, 2001.

[eYD02] Pierre Lombard et Yves Denneulin. Serveur nfs distribué pour grappe de pcs. *RENPAR'14/ASF/SYMPA, Hamamet, Tunisie*, Avril 2002.

[FK90] I. Foster et C. Kesselman. The grid : Blueprint for a new computing infrastructure. *eds. Morgan Kaufmann*, 1990.

[FNPS79] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger et H. Raymond Strong. Extendible hashing - a fast access method for dynamic files. *«ACM Transactions on Database Systems», 4(3) :315-344*, Septembre 1979.

[Gnu] The gnutella protocol specification v0.4.

[GWP99] Bjorn Gronvall, Assar Westerlund et Stephen Pink. The design of a multicast-based distributed file system. *«Operating Systems Design and Implementation»*, pages 251–264, 1999.

[LD02] Pierre Lombard et Yves Denneulin. nfsp : A distributed nfs server for cluster of workstations. Avril 2002.

[LS90] Eliezer Levy et Abraham Silberschatz. Distributed file systems : Concepts and examples. *Département des Sciences de l'Informatique, Université du Texas à Austin, Texas 78712-1188*, 1990.

[MRC+97] Jeanna Neefe Matthews, Drew Roseli, Adam M. Costello, Randy Wang et Tom Anderson. Improving the performance of log structured file systems with adaptive methods. *SOSP 16, St Malo, France*, Octobre 1997.

[NS94] Brian Noble et M. Satyanarayanan. An empirical study of a highly available file system. *«Measurement and Modeling of Computer Systems»*, pages 138–149, 1994.

[OCD+88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson et B. Welch. The sprite network operating system. *«IEEE Computer», pp. 22-35*, Février 1988.

[OD92] John Ousterhout et Fred Douglis. Beating the i/o bottleneck : A case for log-structured file systems. *Département des Sciences de l'Informatique, Techniques des Sciences Electriques et Informatiques, Université de Californie à Berkeley, CA 94720*, Janvier 1992.

[PBB+99] Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steven R. Soltis, David C. Teigland et Matthew T. O'Keefe. A 64 bit, shared disk file system for linux. *Paru dans «the 16th IEEE Mass Storage Systems Symposium held jointly with the 7th NASA Goddard Conference on Mass Storage Systems and Technologies»*, 1999.

[PBB+00] Kenneth W. Preslan, Andrew Barry, Jonathan Brassow, Russel Catalan, Adam Manthei, Erling Nygaard, Set Van Oort, David Teigland, Mike Tilstra et Matthew T. O'Keefe. Implementing journaling in a linux shared disk file system. *Paru dans «the 8th NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with the 7th IEEE Symposium on Mass Storage Systems»*, 2000.

[PSB+00] Brian Pawlowski, Spencer Shelper, Carl Beame, Brent Callaghan et Michael Eisler. The nfs version 4 protocol. *«Engineering Steering Group for consideration as a Proposed Standard in February»*, Février 2000.

[RFC89] NFS : Network file system specification. RFC1094, Mars 1989.

[RFC95] NFS : version 3 protocol specification. RFC1813, Juin 1995.

[RFC97] Hypertext transfet protocol – http/1.1. RFC2616, Janvier 1997.

[RFC00] NFS : version 4 protocol specification. RFC3010, Décembre 2000.

[RO91] Mendel Rosenblum et John K. Ousterhout. Design and implementation of a log-structured file system. *Département des Sciences de l'Informatique, Techniques des Siences Electriques et Informatiques, University de Californie à Berkeley, CA 94720*, Juillet 1991.

[RRT96] Steven R.Soltis, Thomas M. Ruwart et Matthew T.O'Keefe. The global file system. *Paru dans «Proceedings of the 5th NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies»*, Septembre 1996.

[SBMS93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusik et Carl Staelin. An implementation of a log structured file system for unix. *«1993 Winter USENIX - San Diego, CA»*, Janvier 1993.

[Sch96] Roger L.Haskin et Frank B. Schmuck. The tiger shark file system. *Paru dans « Proceedings of the 41st IEEE Computer Society International Conference (COMPCON '96)», pages 226-231, Santa Clara, CA, USA*, Février 1996.

[Sch02] Roger L.Haskin et Frank B. Schmuck. Gpfs : A shared-disk file system for large computing clusters. *Paru dans « Proceedings of the 5th Conference on File and Storage Technologies»*, Janvier 2002.

[SKK+90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E Okasaki, Ellen H. Siegel et David C. Steere. Coda : A highly available file system for a distributed workstation environment. *«IEEE Transactions on computers», Vol 39, Nř4*, Avril 1990.

[slddR01] Georges Da Costa sous la direction d'Olivier Richard. Evaluation de protocoles peer to peer en utilisation à grande échelle : Etude de cas freenet, Septembre 2001.

[SSA+01] H. Stockinger, A. Samar, B. Allcock, I. Foster, K. Holtman et B. Tierney. File and object replication in data grids, 2001.

[SSB+95] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake et C. V. Packer. Beowulf : A parrallel workstation for scientific computation. 1995.

[WA93] Randolph Y. Wang et Thomas E. Anderson. xfs : A wide area mass storage file system. *Département des Sciences de l'Informatique, Université de Californie à Berkeley, CA 94720*, 1993.