NFSP : Une solution de stockage distribué pour architectures grande échelle

Pierre Lombard

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|_/_/_/_/_/_/_/_/_/_/

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : «Informatique : Systèmes et Communications»

préparée au laboratoire Informatique et Distribution
dans le cadre de ***l'École Doctorale «Mathématiques, Sciences et Technologies de l'Information, Informatique»***

présentée et soutenue publiquement

par

Pierre Lombard

le 10 décembre 2003

Titre :

NFSP : Une Solution de Stockage Distribué pour Architectures Grande Échelle

---

Directeur de thèse : Brigitte Plateau

JURY

M. GUY MAZARÉ	, Président
M. PIERRE SENS	, Rapporteur
MME PASCALE VICAT-BLANC/PRIMET	, Rapporteur
MME BRIGITTE PLATEAU	, Directeur de thèse
M. YVES DENNEULIN	, Co-encadrant

La mise en page a été réalisée en grande partie avec LyX, mais certains endroits ont requis une petite goutte de LaTeX. La police utilisée pour le corps du texte est une police avec sérif appelée Palatino. Les schémas ont été réalisés avec `Xfig` *et les courbes, avec* `Gnuplot`*. Le système utilisé pour la rédaction a été constitué d'installations de Debian GNU/Linux.*

Version : 2003-12-15 19:03

"It is a mistake to think you can solve any major problems just with potatoes."

Douglas Adams

Remerciements

Cette thèse s'est déroulée au sein du laboratoire Informatique et Distribution (ID-IMAG). Grâce à un financement BDI CNRS, j'ai eu l'opportunité de pouvoir développer ce projet pendant trois années, mais aussi celle de pouvoir interagir avec des gens aux compétences pointues dans un domaine technique très intéressant.

Maintenant que cette période touche à sa fin, je dois tout naturellement adresser mes remerciements à de nombreuses personnes :

Tout d'abord à **Yves Denneulin**, mon co-directeur de thèse pour nombre de choses, et entre autres parce qu'il a aussi su orienter mon travail quand il le fallait et a dû (et su) supporter mes interrogations et mes nombreuses questions pendant toutes ces années,

à **Brigitte Plateau**, qui m'a accueilli dans son laboratoire et qui aussi été mon directeur de thèse, pour l'encadrement efficace qu'elle a su réaliser, ainsi que pour les opportunités dont elle m'a permis de bénéficier,

à **Pierre Sens** et **Pascale Vicat-Blanc/Primet**, pour avoir bien voulu accepter la charge de rapporter cette thèse et pour leurs remarques constructives qui m'ont permis d'améliorer ce mémoire,

à **Guy Mazaré** qui a bien voulu m'accorder l'honneur d'être le président de mon jury,

à **Adrien Lebre**, **Olivier Valentin**, **Christian Guinet**, **Rafael Ávila** et **Olivier Lobry** pour tout ce qu'ils ont permis de mettre en place et de réaliser,

à **Bruno Richard**, pour ces nombreuses discussions que nous avons pu avoir lors de ces trois années de partage de bureau,

à **Cyrille Martin** et **Olivier Richard** pour la mise au point de la grande théorie unifiée de la TP-complétude,

à **Stéphane Martin,** pour avoir toléré avec tant de calme que je casse son précieux ^W^Wsa grappe,

à **Saïd Oulahal**, **Mauricio Pillón**, **Jésus Verduzco** pour les nombreuses conversations partagées,

aux membres du **Laboratoire ID-IMAG** (permanents, assistantes, ingénieurs, thésards, stagiaires) pour l'excellente ambiance de travail (mais pas uniquement !) qui n'a jamais cessé d'y régner pendant ces quelques années,

à tous mes amis qui m'ont supporté et encouragé,

et à ma famille, et plus particulièrement mes frères, ma grand-mère et mes parents pour tant de choses.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Le développement des architectures de grappes utilisant des ordinateurs personnels est un phénomène qui ne cesse de gagner en ampleur depuis maintenant de nombreuses années. Très souvent, ces machines utilisent des systèmes d'exploitations libres dérivés d'UNIX. En effet, s'ajoutant à leur faible coût de licence, la disponibilité du code source permet de mettre en place de nombreuses expériences à tous niveaux.

Ces ordinateurs personnels identiques à ceux utilisés par le grand public sont composés d'un processeur, de mémoire, d'un disque dur et d'une carte réseau. Ces grappes de machines lorsqu'elles sont équipées d'un système d'exploitation Linux sont souvent appelées grappes (ou *clusters*) Beowulf[1] [SSB+95]. Elles permettent maintenant depuis plusieurs années d'obtenir un rapport puissance/prix très avantageux lorsqu'il est comparé aux solutions classiques utilisant de puissantes machines dédiées telles que les super-calculateurs et se retrouvent très souvent dans le classement du TOP500 des 500 machines les plus puissantes du monde (selon un critère calculatoire).

L'architecture différant entre les super-calculateurs et les grappes, les modèles de programmation optimaux sont distincts. Sur super-calculateurs (gros systèmes multi-processeurs), le paradigme par mémoire partagée est le mode de fonctionnement naturel mais il s'avère bien souvent inefficace (sans matériel dédié) sur architecture de grappes. Sur ces dernières, le mode de communication par passage de messages réseau s'avère le plus naturel afin de partager des informations entre deux processeurs.

Si l'on considère le TOP500, il s'avère rare de trouver une grappe Beowulf utilisant des composants très standards mais, dans le cadre du projet *i-cluster*, cette grappe de 225 nœuds complètement standards et non « dopés » constitués d'un réseau ethernet à 100Mb/s et de simples disques IDE de stations de travail a pu finir classée 385ème en 2001 [RAM+01].

Ce classement calculatoire met en jeu principalement deux composants des nœuds à savoir la mémoire et le processeur, mais il reste un composant non utilisé : le disque

[1] En référence au héros légendaire du poème épique éponyme composé au VIIIème siècle, qui aurait tué un super-calculateur nommé Grendel, ou bien peut-être était-ce un dragon. La légende est peu claire à ce sujet 8-)

dur. Usuellement, ceux-ci sont principalement utilisés pour stocker le système d'exploitation du démarrage ainsi que des fichiers temporaires mais, en revanche, peu de systèmes proposent d'utiliser cet espace pour effectuer du stockage de données, ce qui constitue de l'espace inutilisable pour les utilisateurs.

D'un point de vue architecture, la grappe constitue une entité relativement homogène pour plusieurs raisons : une raison pratique étant que les machines sont achetées en même temps (!), et une autre que nombre d'algorithmes parallèles sont plus confortablement déployés sur des grappes homogènes. Ainsi, puisque ce type d'architectures permet de remplacer avantageusement (bien sûr selon le type des problèmes considérés) un super-calculateur, l'idée de mettre en relation de tels ensembles de machines a vite germé, et a ainsi donné naissance aux premières « grappes de grappes » et donc rajouté un niveau d'architecture.

Ce concept est très proche du concept des « grilles » de machines [FK99] mais offre une approche plus réalisable. En effet, le modèle correspondant à celui d'un réseau électrique (*power grid*), à savoir « *je branche la prise et ça marche* », se heurte à quelques difficultés de mise en œuvre[2]. En effet, de même que les différents pays ont des formes de prise, des tensions et des fréquences de courant différentes, le chemin vers la mise en relation de manière simple et efficace des divers constituants d'une grille de machines est pavé de nombreux obstacles tels que : multiplicité des « zones d'autorité » (administrateurs, institutions), confidentialité des données, authentification, sécurité, diversité des architectures matérielles et logicielles, ...

C'est ainsi que sont apparus plusieurs « parfums de grille », s'orientant alors plus particulièrement sur la résolution de problèmes tels que la gestion des données avec le projet européen Datagrid ou bien sur les applications et la puissance de calculs. Nous évoquons en effet le projet DataGrid car celui-ci est né des besoins de stockage des physiciens, besoins caractérisés par un volume énorme de données en perpétuelle croissance. Dans le cas des expériences et des simulations de physique de particules à haute énergie, les quantités de données sont énormes et doivent atteindre des péta-octets de données par an ! De telles quantités soulèvent le problème de leur transport : la bande passante effective d'une camionnette remplie de systèmes de stockages est, certes, très élevée mais les défauts de cette approche sont que, d'une part la latence est élevée, pas forcément garantie et que d'autre part, le système n'est en définitive pas très souple.

C'est donc logiquement que des expériences de transferts sur réseaux dédiés à haut débit ont commencé à se développer depuis maintenant plusieurs années. Les organismes de recherche tel que l'INRIA ont commencé à développer des collaborations avec plusieurs partenaires afin de mettre en place des expérimentations sur les réseaux à haut débit dans le cadre des projets VTHD (*Vraiment Très Haut Débit*) avec des supports physiques supportant jusqu'à 10Gb/s.

Néanmoins, les équipements des sites connectés (routeurs) ne peuvent actuellement utiliser qu'une partie de cette bande passante pour des raisons de coût et du peu de

[2]Sauf pour quelques « bonnes » applications dont la réalisation ne requiert que très peu de synchronisations avec un serveur central comme par exemple Seti@home, les divers projets de cassage de clés (RC5DES, ...) ou bien pour les simulations à base de méthodes de Monte-Carlo.

choix de routeurs supportant de tels flux. Le problème qui se pose alors dans ce cadre est d'obtenir des possibilités de transferts efficaces entre deux sites puis, dans un second temps entre plusieurs sites.

Une autre question qui se pose est que la plupart des sites n'a pas de matériels homogènes ni des performances équivalentes et que, comme nous l'avons précédemment évoqué à propos de la mise en place des grilles de machines, les problèmes qui se posent ne sont peut-être pas très difficiles à résoudre mais leur nombre les rend très consommateurs de temps. Aussi, l'utilisation de systèmes légers de stockage, peu intrusifs, standards et ne remettant pas en cause les architectures logicielles et matérielles d'un site peut-elle permettre d'offrir une solution utilisable pour gérer un stockage distribué sur une grappe. De plus, cette approche constitue aussi une possibilité pour offrir une solution de transfert efficace entre sites distants de plusieurs centaines de kilomètres, en pouvant remplir le « tuyau » réseau grâce à l'agrégation des disques et des cartes réseau « grand public ».

Pour résumer le contexte, il y a donc plusieurs problématiques autour de ce domaine : offrir du stockage en utilisant du matériel standard constituant des grappes de PC, tout en permettant d'avoir de bonnes performances dans le cadre de transferts de données sur grilles. C'est ainsi que nous avons choisi de nous consacrer à ces problèmes très actuels. Ce mémoire va donc s'efforcer de fournir un aperçu des recherches et des expérimentations que nous avons été amenés à effectuer afin de contribuer à ce domaine.

Cet aspect de performance des systèmes d'information ayant commencé à se développer très tôt, les documents et les systèmes traitant de ce sujet sont très nombreux. Afin de bien expliciter les concepts sous-jacents à ce domaine et les différentes questions soulevées, nous avons choisi de décrire dans le chapitre 2 page 21 de manière un peu plus détaillée les problèmes qui se posent lorsque l'on évoque l'accès aux données en prenant pour exemple un système de fichiers réel et standard du monde Linux, le système EXT2. Cet exemple, certes simple, a le mérite de permettre de mieux cerner tous les contraintes qui se posent lors de la mise au point de nouveaux systèmes ou bien lors de la recherche d'un système adapté aux besoins des utilisateurs. C'est ainsi que cette partie complémentaire de l'introduction, mais néanmoins séparée car un peu plus technique, nous permettra d'établir une liste de critères que nous observerons sur plusieurs solutions existantes.

Nous présenterons ensuite deux grandes familles de solutions de stockage distribué par bibliothèques de programmation (c'est-à-dire au niveau applicatif) dans le chapitre 3 (page 31) puis les solutions utilisant un système de fichiers dans le chapitre 4 (page 43).

Nous justifierons ensuite notre proposition de système de stockage dans le chapitre 5 (page 71) puis nous décrirons les implantations des prototypes réalisés ainsi que les résultats obtenus dans le chapitre 6 (page 83). Dans le chapitre suivant (chapitre 7, page 111), nous présenterons les diverses expérimentations envisagées afin d'améliorer le fonctionnement des prototypes au niveau de leurs performances et de la sûreté du stockage.

Finalement, nous présenterons plusieurs applications qui ont pu être construites au dessus des prototypes réalisés dans le chapitre 8 page 125, la première étant une application de transfert à haut débit entre deux grappes Beowulf (GXfer) développée dans le cadre du projet RNTL E-Toile, et l'autre, NFSG permettant de mettre en place un système de stockage distribué sur grappes en utilisant GXfer et s'interfaçant grâce à NFS.

Finalement, nous conclurons ce mémoire dans le chapitre 9 page 135 en dressant un récapitulatif des différentes contributions effectuées ainsi qu'un bilan des approches que nous avons été amené à développer. Nous terminerons en évoquant diverses pistes de travaux futurs qui peuvent être empruntées afin de poursuivre dans les voies tracées.

Chapitre 2

Le stockage des données

Ce chapitre a pour buts principaux d'exposer de manière générale les divers concepts que nous allons développer au cours de ce document. Nous évoquerons aussi succinctement une partie des problèmes que la manipulation de ces données peut poser par la suite. Nous présenterons aussi plusieurs des points que nous avons jugés importants et qui ont particulièrement guidé la réalisation du travail exposé dans la suite de ce document.

2.1 Une vision simplifiée du stockage

Si l'on effectue une comparaison du fonctionnement des systèmes de fichiers avec le fonctionnement d'un pile de protocole réseau de nombreux points communs apparaissent entre les deux domaines :

- Au niveau le plus bas, se situe le support physique, ce peut par exemple être (et ce sera souvent) un support magnétique (disque dur).
- Au dessus de ce niveau se situe une vue support logique, qui abstrait d'une certaine façon la couche basse : pour un disque dur ce rôle peut être assuré par le contrôleur d'interface et ainsi offrir aux couches supérieures un vision linéarisée du disque dur. Dans le cadre d'un réseau local, ce niveau pourrait par exemple être vu comme le protocole de communication Ethernet.
- Sur ces couches, d'autres couches enrichissant à chaque fois le modèle peuvent être ajoutées : alors que pour le réseau des protocoles tels que IP/TCP sont ajoutés, pour le stockage ce sera par exemple un système de fichier ou bien un système d'agrégation de disques (techniques RAID) sur lequel un système de fichier pourra être mis en place. Dans tous les cas, des couches de plus complexes peuvent être ajoutées afin d'apporter de plus en plus de services.

Puisque nous nous intéressons plus particulièrement aux systèmes de stockage et aux moyens d'accéder aux informations, nous allons plus orienter notre discours sur ce thème. Afin d'illustrer nos propos, nous allons par la suite considérer un disque dur

classique reliée à un ordinateur personnel. Comme nous venons de le présenter, ce périphérique de stockage offre plusieurs couches visibles au système :

- la couche physique : celle-ci peut être spécifique à un fabricant ou bien à un modèle de disque
- la couche logique : l'aspect logique est assuré par le contrôleur du disque, celui-ci va alors fournir une interface standard qui va permettre aux couches supérieures de faire plus ou moins abstractions du fonctionnement du disque. Par exemple, il existe des « disques » IDE (le format usuel des disques durs pour PC) qui ne gèrent pas un stockage sur support magnétique mais dans des systèmes de RAM Flash (dans les systèmes embarqués notamment).
- une couche utilisable pour le système d'exploitation : c'est à ce niveau là que va se situer le système de fichiers. Cette couche correspond grossièrement dans les systèmes Unix à la couche *block device* ou, périphérique en mode bloc.
- une couche utilisable pour l'application : c'est elle qui va offrir les maintenant classiques hiérarchies de dossiers et de fichiers.

La figure 2.1 représente l'empilement de couches (quelque peu simplifié) qui peut être observé dans un système Linux.

Le système de fichiers va permettre de multiplexer sur un média « linéaire » (vu sous la forme d'une suite de blocs) différentes sources d'informations qui seront usuellement appelées fichiers.

Afin de pouvoir mettre en place une structure arborescente, certains fichiers seront dits non-réguliers car ils ne contiendront pas de données comme un fichier standard mais plutôt des listes de fichiers : ce seront les répertoires. D'autres fichiers peuvent aussi avoir une signification particulière pour les systèmes d'exploitation : sous UNIX ce peuvent être par exemple des fichiers de périphériques *(devices)* qui correspondent à des portes d'entrées bien spécifiques du système, sous les environnements Microsoft, certains fichiers spéciaux demeurent tels que `CON` ou `AUX` et sont principalement des héritages du passé.

Nous avons donc brossé un schéma général (et simplifié) du fonctionnement du stockage, nous allons nous intéresser plus particulièrement à la couche système de fichiers.

2.2 Des données et des méta-données

Si l'on regarde d'un peu plus près ce qui constitue un fichier (nous restons pour l'instant dans le cadre système local), il y a évidemment les données contenues dans le fichier, c'est-à-dire la source de l'information. Cependant, cela peut ne pas apparaître de prime abord mais pour pouvoir gérer plusieurs données sans les mélanger, il est nécessaire de maintenir des informations sur ces données et c'est à cette tâche que servent les méta-données.

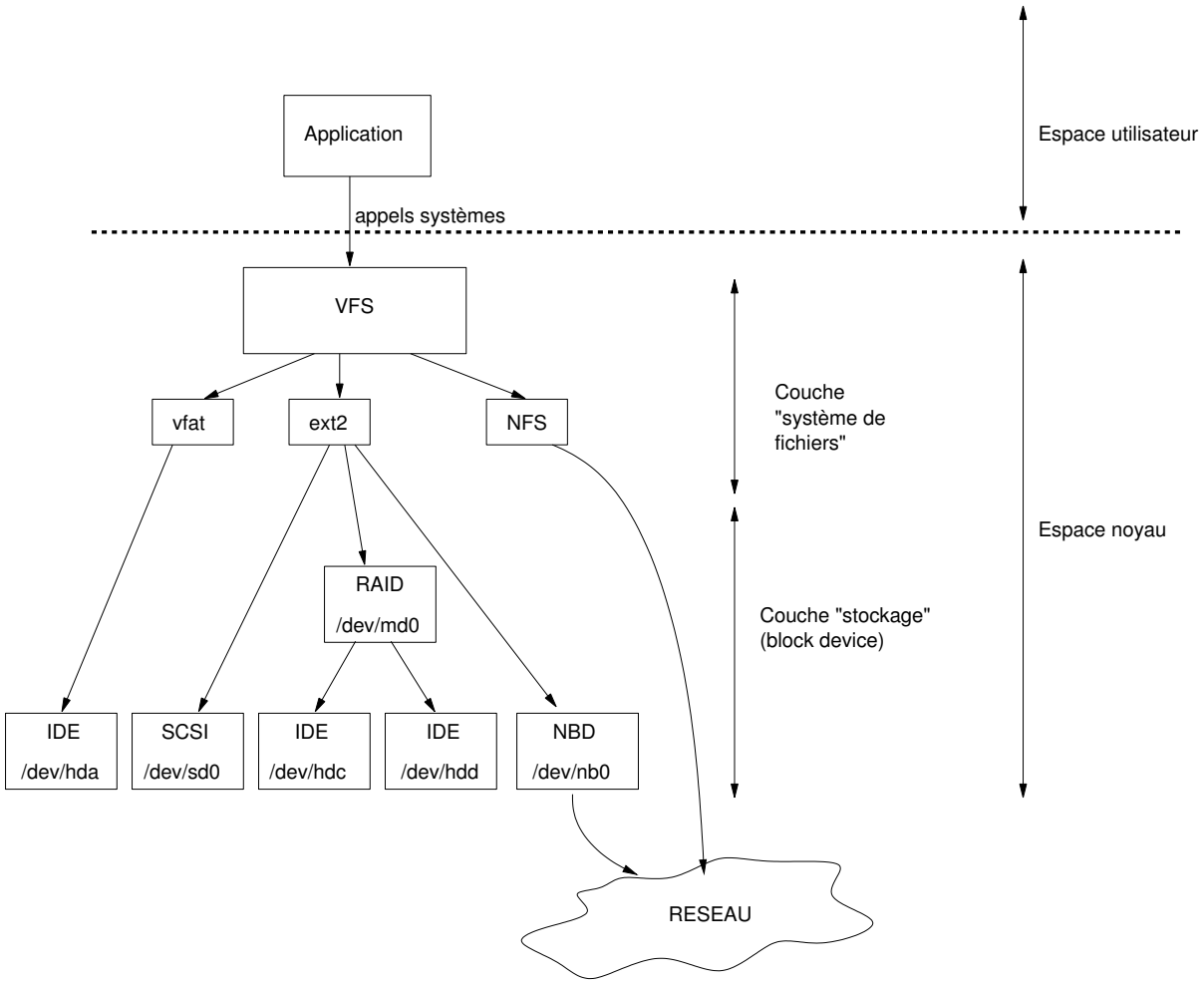

FIG. 2.1 – Couches de stockage

Les méta-données peuvent être de différentes natures et contenir à peu près tout et n'importe quoi :

- elles contiennent au moins assez d'informations pour pouvoir retrouver les données. Dans le cas simple d'un fichier, ce peut simplement être un bloc de départ et une taille. Cependant si l'on considère un système actuel, on risque plutôt d'avoir des structures complexes décrivant les blocs de données vraiment utilisés, voire partagés entre plusieurs fichiers, ou bien des spécificités par morceaux de fichiers, ...
- la taille du fichier est aussi gérée à ce niveau : par exemple un fichier « creux » (c'est-à-dire un fichier avec seulement quelques blocs non-nuls) peut avoir une taille pour l'application qui sera différente de la taille qu'il occupera physiquement sur la couche de stockage (en effet il est inutile de stocker des blocs complètement remplis de zéro puisque, si les données ne sont pas là, le système doit retourner des zéros).
- elles peuvent contenir des informations additionnelles permettant de gérer des droits et des propriétés sur les données, des sommes de contrôles, des signatures, une indication sur la provenance des données ou un commentaire, un numéro de version, ... L'étendue des informations stockables n'a à priori pas de limite et est plutôt dicté par les souhaits des utilisateurs.
- souvent des données statistiques/historiques sont présentes (dates de création, de dernier accès, de modifications, ...) En plus de permettre d'avoir une idée sur l'utilisation des fichiers, certains systèmes peuvent avoir tendance à regrouper les fichiers fréquemment utilisés dans des endroits du disque ayant un meilleur temps d'accès (utilitaires de défragmentation et d'optimisation des disques).

Étant donné la grande diversité du contenu de ces méta-données, toutes ne vont pas avoir le même impact sur l'accès aux données. Alors que certaines peuvent être considérées cruciales pour l'accès au contenu (localisation des données, taille, etc...) d'autres peuvent apparaître comme secondaires telles que la date de dernier accès au fichier (cela dépend bien sûr de ce que l'utilisateur attend).

Pour illustrer ces propos, nous allons prendre pour exemple le fonctionnement d'un système de fichiers classique sous Linux, le système EXT2[CDM97]. Ce système a été originalement développé par Rémy Card et s'inspire des travaux précédemment menés dans les différentes versions et « parfums » d'UNIX (système Unix [RT78] et BSD [MJLF84] principalement). Dans sa version standard, il n'offre pas toutes les extensions des systèmes plus récents telles que la journalisation ou bien une gestion optimisée des gros répertoires... Cependant, le fait qu'il soit le standard *de facto* depuis plusieurs années a fait que le développement d'extensions pour résoudre ces limitations est toujours actuel.

Nous considérerons ici la version classique d'EXT2 car elle permet d'appréhender de manière plus aisée les différents problèmes qui peuvent se poser lorsque l'on développe un système de fichiers.

La figure 2.2 illustre la manière dont le système de fichier découpe l'espace linéaire du disque en sous-ensembles dans lequel il va gérer les fichiers. Au début du disque est réservée une partie qui peut être utilisée pour *booter* le système puis le disque est découpé en divers sous-ensembles de blocs. Ce découpage permet par exemple de

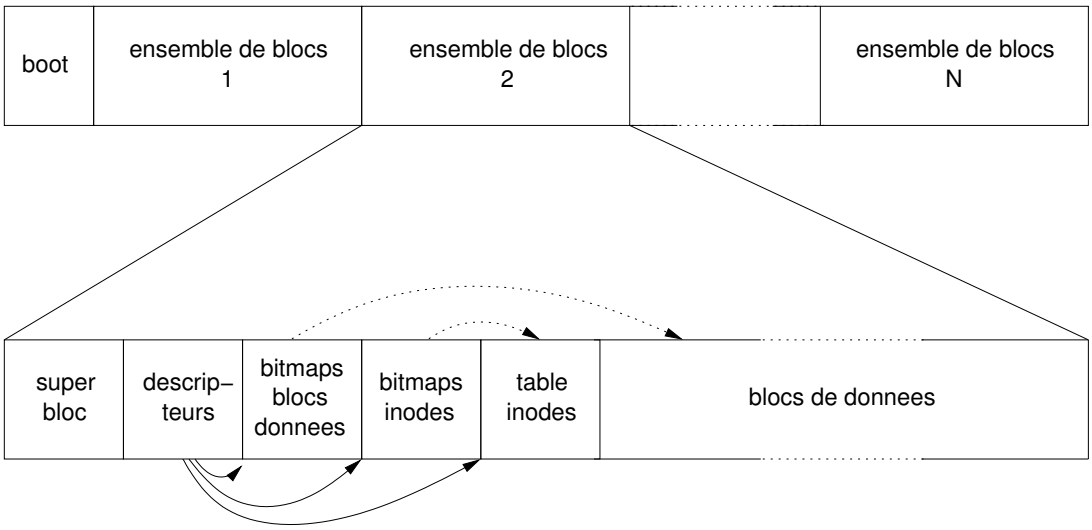

FIG. 2.2 – Architecture générale EXT2

regrouper les fichiers ayant de fortes affinités (par exemple un répertoire et ses fichiers) dans une même zone du disque, et ce afin de limiter les mouvements de la tête de lecture/écriture du disque.

Chaque ensemble de bloc commence par un super-bloc, celui-ci étant un point fixe, il va permettre de spécifier où les autres zones critiques du système se trouvent (*bitmaps* d'inodes, de blocs, inodes, ...) Ainsi, la table d'inodes sera réservée à la création du système et comprendra les différentes informations sur les fichiers qui seront stockés dans la zone « blocs de données ». Pour gérer la localisation des données associées à un inode, un système de redirection similaire à ceux des premiers UNIX est utilisé et une partie de ces informations de localisations est située dans la structure même de l'inode. Ainsi, les petits fichiers ont la liste de leurs blocs directement dans l'inode et les gros fichiers requièrent l'allocation d'un nouveau bloc pour pouvoir stocker leurs méta-données de localisation. Le principe de fonctionnement des blocs d'indirection a été mis en place au sein du premier Unix [RT78] et est illustré sur la figure 2.3.

En partant de cet exemple concret, les divers points suivants peuvent être remarqués concernant ce système :

– les méta-données étant elles-mêmes des données, elles pourraient aussi être stockées comme des fichiers et on aurait alors des méta-méta-données[1]
– les méta-données sont usuellement de plusieurs ordres de magnitude plus petites que les données auxquelles elles sont associées. Toutefois, selon le type d'utilisation du système (par exemple un serveur de courrier), il peut n'y avoir que de très petits fichiers et les méta-données peuvent alors avoir une taille non-négligeable.
– les méta-données sont très souvent consultées donc il est mieux de pouvoir les sto-

[1] L'utilisation (et l'intérêt) d'utiliser des méta-méta-...-méta-données est laissé à l'imagination du lecteur 8-)

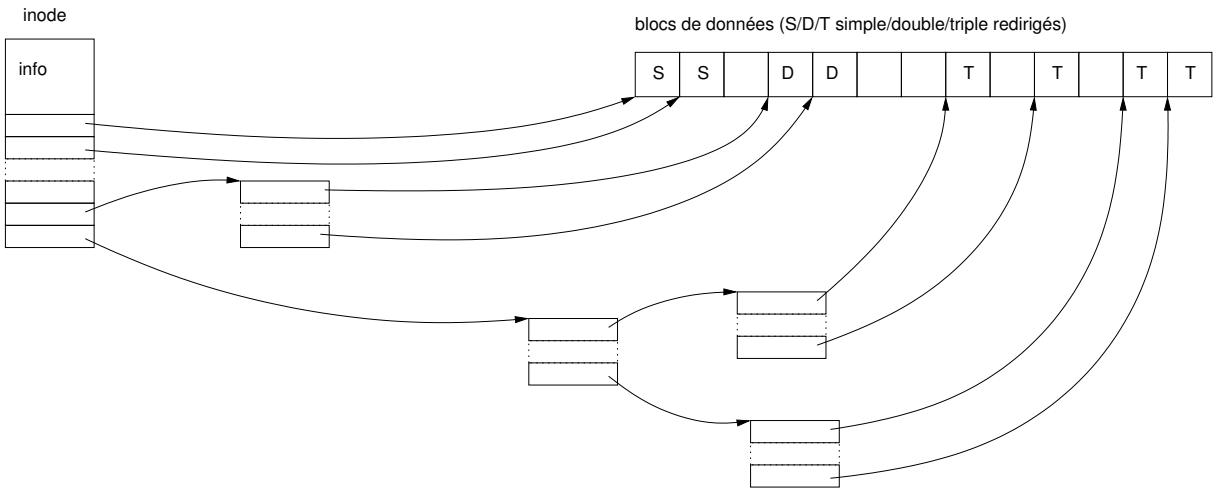

FIG. 2.3 – Liens entre inode et blocs de données pour un fichier régulier en EXT2

cker sur un support rapide (dans un cache mémoire par exemple). Certains systèmes de fichiers locaux actuellement développés (par exemple *ConquestFS* [Wan03]) vont même jusqu'à les stocker uniquement dans des mémoires non-volatiles (RAM alimentées), les données allant pour leur part sur support magnétique classique.

À cette gestion mettant en œuvre des données et méta-données, se superpose un système de nommage qui permet usuellement de faire correspondre les méta-données avec un identifiant alpha-numérique. Ainsi dans `EXT2`, en partant d'un répertoire racine « / », stocké comme un fichier spécial (un répertoire) contenant la liste des fichiers et sous répertoires qu'il contient, le système peut accéder à l'arborescence. Les étiquettes « collées » sur les méta-données sont en fait gérées par les répertoires (une sorte de fichiers spéciaux) qui permettent de maintenir la correspondance inode-nom.

Ce système partant systématiquement de la racine « / » est appelé VFS (*Virtual File System*) car il permet de mélanger au sein d'une même arborescence « virtuelle » plusieurs systèmes de fichiers. Par exemple, dans les environnements UNIX, une arborescence peut contenir des répertoires gérés en local (systèmes locaux), d'autres gérés sur un serveur distant (par réseau), ou bien un mélange de répertoires local et distant (comme le `unionfs` des BSD)... Le principe de cette approche est de déléguer la gestion d'un répertoire à un autre système par le biais d'un montage (procédé réalisé par l'appel système `mount`). L'avantage de cette approche est de fournir au sein d'une machine un moyen uniforme d'accéder aux fichiers sans avoir à se soucier des mécanismes sous-jacents utilisés.

Plusieurs problèmes sont donc présents lorsque l'on commence à regarder les systèmes assurant une gestion des données :

– la gestion du système de nommage ou, comment à partir du nom d'une ressource obtenir une référence sur celle-ci (ou, comment trouver la méta-donnée) ?
– comment être sûr qu'un nom de ressource sera unique au sein du système ?

– à partir de la méta-donnée, comment accéder aux données ?
– si l'on se place dans un cadre distribué, comment accéder aux méta-données et aux données, et ce, si possible, de manière efficace ?

Un système simple de nommage peut être fourni par les URIs (*Universal Resource Identifier*) qui indiquent complètement la localisation de la donnée en utilisant un nom du type `protocole://machine/répertoire/fichier`. Cette adresse peut alors représenter l'accès aux méta-données ou bien aux données, les combinaisons possibles étant elles mêmes dépendantes du protocole utilisé.

En ce qui concerne les derniers points abordés, plusieurs niveaux de distribution peuvent être envisagés sur une architecture dédiée de stockage (SAN), sur des machines « dopées » reliées par réseau à haut débit, faible latence, sur des « super-calculateurs du pauvre » (grappes de PCs), sur une architecture de grille ou bien de grappes de grappes, ... Bien évidemment, toutes ces architectures très différentes ne vont pas offrir les mêmes services et performances pour des raisons de prix et/ou de contraintes physiques, aussi les contraintes (modèles de cohérence) qui vont être imposées sur les données et les méta-données risquent-elles de devoir être adaptées au support matériel.

Alors que dans un cadre local, le couplage des données et de leur méta-données peut être fort, voire quasiment synchrone (toute modification de l'une peut immédiatement influer l'autre), le passage à un cadre distribué ou même faisant simplement intervenir des accès réseaux expose toute une nouvelle branche de problèmes de gestion de la cohérence. En effet, un couplage fort a peu de chance d'être utilisable car il risque de faire s'effondrer les performances du système par le nombre accru de synchronisations...

Cependant, si l'on observe un système géré localement, et malgré la latence relativement faible des accès à un disque local, il peut y avoir un faible découplage entre les méta-données et les données (sauf pour certains systèmes journalisant ces deux éléments). En effet, si l'on considère la majorité des systèmes journalisés actuels disponibles pour la plateforme Linux, ceux-ci ne journalisent que les méta-données, ce qui signifie qu'en cas d'interruption inopinée du système, seules celles-ci seront cohérentes, les données seront alors plus ou moins cohérentes selon la gravité de la panne... Ceci illustre principalement le fait que la gestion des données et des méta-données peut être quelque peu découplée, le problème principal à résoudre étant de fournir une sémantique acceptable à l'utilisateur.

2.3 Standards

Un autre aspect du stockage est peut être vu comme une réponse à : « Comment accéder à mes données ? » La réponse à cette question est triviale sur un système d'exploitation local puisque celui-ci est normalement livré avec son propre système de fichiers. En revanche, si le système doit utiliser des données venant d'un système différent, cela peut poser de nombreux problèmes s'il n'est pas déjà supporté par le système natif car,

sauf conception identique, il y a très peu de chances de pouvoir utiliser un type de système de fichiers avec un autre. Le système d'exploitation peut supporter plusieurs types de fichiers bien sûr, mais le plus simple est souvent de faire un transfert des données par le biais d'un réseau, ce qui permet aux deux machines de communiquer sans problèmes moyennant un accord sur le protocole à utiliser.

En effet les normes réseaux telles que « TCP/IP » par exemple permettront aux deux machines de communiquer dans la mesure où les implantations respecteront une norme définie (par l'IETF pour ce cas). Les modifications à apporter aux deux systèmes seront minimales puisque tout sera déjà disponible pour l'échange des données.

Pour revenir aux systèmes de fichiers « locaux », il n'y a pas vraiment de système qui se détache ou bien qui soit normalisé. La plupart des systèmes s'engagent à respecter une API (POSIX par exemple) mais compte tenu de la variété des données, des métadonnées et de leur utilisation envisagée, il y a rarement compatibilité entre plusieurs systèmes. Comme contre-exemple, nous pouvons penser aux CD-ROM qui, pour la plupart, sont gravés selon la norme ISO9660 et peuvent être lus sur la plupart des systèmes actuels. Même si toutes les extensions ne sont pas supportées (*Rock Ridge* ou *Joliet*), un mode de compatibilité minimal est disponible (niveau 1).

Ainsi, de notre point de vue, le respect des standards est un point très important car il permet à moindre coût de relier des architectures très différentes sans avoir de coût supplémentaire de développement. Certes, cette approche peut s'avérer plus contraignante par le respect imposé de la norme et peut aussi limiter le développement des implantations à effectuer, mais elle permet d'offrir une intrusivité minimale et de conserver les systèmes fonctionnant déjà bien, sans les modifier.

D'autre part, un aspect pratique à considérer, si l'on administre un parc de machines ayant un support vendeur, est que le chargement d'extensions non garanties dans le système peut faire perdre le support du vendeur. Charger de telles extensions peut aussi causer des problèmes de *crash* du système si elles n'ont pas été suffisamment *débuggées*, ce qui peut être plus ou moins tolérable selon le type de parc envisagé : pour une grappe expérimentale, ce peut juste être un ennui pour l'administrateur alors que pour une plus grosse grappe de production, ce peut être inenvisageable.

Ainsi, l'utilisation de protocoles standardisés peut limiter les solutions à envisager mais ce désavantage est fortement compensé par la faible intrusivité des solutions apportées.

2.4 Conclusion et critères d'évaluation

Dans la suite de ce document, nous allons donc nous efforcer d'étudier diverses solutions développées pour gérer un composant de stockage qui pourrait satisfaire nos besoins. Nous avons retenu plusieurs critères et ceux-ci sont synthétisés dans le tableau 2.1.

Nous avons retenu trois grandes familles de critères :

– critères environnementaux (« Dans quel environnement logiciel et matériel, le système est-il disponible ? »)
– critères de fonctionnalités disponibles (« Quelles sont les fonctionnalités et les spécificités disponibles ? »)
– critères de pérennité (« Est-ce que le système peut évoluer facilement ? »)

L'environnement regroupe les différentes implications sur l'architecture logicielle ou matérielle requise pour faire fonctionner un système : moins il y a de dépendances, et évidemment mieux c'est ! En effet, un système très dépendant du matériel risque par exemple de ne pas pouvoir évoluer facilement, si les périphériques évoluent ou changent. S'il fonctionne de plus sur un système « exotique », il peut de plus ne pas être bien testé et cet aspect rapproche cette solution d'un aspect pratique mais néanmoins à prendre en compte, à savoir la robustesse du système. En effet, peu d'utilisateurs souhaitent utiliser un système qui n'a pas vraiment fait ses preuves, ce qui certes a tendance à produire un cercle vicieux... Ces critères ont, certes, un aspect pratique, mais cet aspect nous semble important lors de la conception d'une nouvelle architecture pour qu'elle puisse d'une part avoir un certain retour, et éventuellement devenir « populaire ».

La seconde famille de critères regroupe les fonctionnalités du système. Par exemple, pour l'une des applications que nous envisageons (transfert parallèle haut débit site à site), il est nécessaire que les données d'un fichier soient distribuées afin que cette opération puisse être optimisée. Il est aussi nécessaire d'avoir une API permettant d'accéder aux données, car sans cela des modes évolués d'accès aux informations, différents de ceux imaginés par les concepteurs originaux ne pourront pas être envisagés sans de nombreuses modifications... ce qui quelque part, a tendance à diminuer fortement l'intérêt expérimental (et donc scientifique) de l'utilisation possible de ce système.

Finalement, la dernière grande famille de critères est un peu, voire beaucoup, plus pragmatique et concerne la pérennité du système. En effet, l'utilisation d'un système standardisé et ouvert a plus de chances de survivre au temps qu'un système fermé, dans la mesure où il peut être maintenu, même si la société ou les individus qui le fournissent cessent leur activité. Dans ce cas particulier, les utilisateurs ne vont avoir d'autre choix de transférer leurs données sur un autre système pour peu que ce problème ait été prévu à la base. Sans cela, et avec un système « exotique », les données risquent fort, d'être perdues, ce qui quelque part diminue très fortement l'intérêt du stockage et a tendance à bien prendre en compte un certain adage disant que « stocker les données c'est bien, mais pouvoir y accéder, c'est mieux » :-)

Nous avons donc passé plusieurs des moyens de gestion et d'accès aux donnés sous ce crible et nous allons présenter dans les chapitres 3 page 31 et 4 page 43 les résultats de nos observations. Le chapitre 3 traitera plus particulièrement des systèmes d'accès aux données par le biais de bibliothèques spécifiques à lier aux applications ou d'outils dédiés. Le chapitre 4 se contentera plus pour sa part de traiter le cas des systèmes intégrés aux systèmes d'exploitations. Ces systèmes intégrés permettent en effet de voir ses applications fonctionner de manière native grâce à la disponibilité du VFS et de l'API POSIX (`open()`,`read()`,`write()`,`close()`,...)

Famille	Critère	Description
Environnement	Intrusivité	Ce qui doit être modifié par rapport à une installation standard du système : 0 : rien/application indépendante + : applications dédiées ou re-*linkage* des applications ++ : modification du système
	Pré-requis	Besoins matériels spécifiques : 0 : machines standards + : serveurs dédiés +++ : matériel dédié et/ou spécifique
	Support Linux (Linux)	Composants disponible pour une architecture Linux (grappes *Beowulf*) : 0 : rien C : client (ou équivalent client) S : serveur(s) (ou équivalent serveur) * : tout (tout est disponible)
	Interface (API)	Possibilité de développement d'applications tierces : sp. : utilisation d'une API spécifique (re-compilation d'applications) VFS : API standard POSIX (applications standards)
Fonctionnalités	Distribution des données (D)	Où les données sont elles situées ? 0 : centralisées : toutes les données sur une machine + : intermédiaire - distribution à gros grains (répertoire/fichiers/volumes) ++ : totale - distribution à grain fin (morceaux de fichiers - *striping*)
	Distribution des méta-données (MD)	Où les méta-données sont elles situées ? 0 : centralisées : toutes les méta-données sur une machine + : intermédiaire - distribution à gros-grain (répertoire) ++ : totale - distribution à grain fin (fichiers)
	Cohérence	Si des informations sont cachées, quel est le niveau de cohérence fourni : 0 : pas de cache + : cohérence faible (sémantique temporelle) ++ : moyenne (sémantique session) +++ : forte (sémantique POSIX)
Pérennité	État	Le système est-il toujours développé/supporté ? - : non + : oui ++ : oui et développement actif ? : pas d'information
	Licence	Quelles sont les contraintes « légales » d'utilisation ? standard (S) / ouvert (O) / fermé (F) / propriétaire (P) / non-explicite (?)

TAB. 2.1 – Nos critères d'évaluation

Chapitre 3

Comment accéder aux données

Alors que le premier chapitre avait pour but de faire une présentation rapide des différents concepts reliés à la gestion des données, notamment la séparation entre les données et les méta-données, cette partie s'intéresse plus particulièrement aux systèmes existants et disponibles pour gérer des données. Nous ne nous intéresserons pas uniquement aux systèmes de fichiers (c'est à dire complètement intégrés dans le VFS) mais d'une manière plus générale aux divers moyens de stocker et d'accéder à des données.

3.1 Introduction

Pour gérer des données, il existe plusieurs grandes familles de techniques et bien évidemment, les limites et les contours de ces ensembles peuvent être plus ou moins nets en fonction des choix des concepteurs des systèmes. Nous allons définir les systèmes principaux permettant de gérer les données :

- Les systèmes de gestion de base de données : Ces systèmes permettent bien évidemment de gérer les données mais sont dans une branche tout à fait à part, aussi ne nous attarderons-nous pas plus sur ce domaine. Toutefois, de nombreuses techniques dans ce domaine « transpirent » dans les systèmes de fichiers locaux (journalisation, gestion de transactions, utilisations de structures d'index évoluées pour optimiser la recherche des fichiers, ...)
- Les bibliothèques d'accès à des systèmes de stockage : Ces systèmes d'accès requièrent que l'application soit recompilée pour tirer partie de leurs avantages. Une intégration avec des API un peu plus standard peut être envisagée comme par exemple avec MPI-IO, mais, la plupart du temps, une sémantique et une API POSIX peuvent difficilement être obtenues. Nous présenterons quelques unes de ces bibliothèques ainsi que les protocoles utilisés.
- Les systèmes de transfert : Contrairement aux bibliothèques évoquées dans le point précédent, ces systèmes sont plutôt utilisés dans l'optique de mettre en place un système de pré-chargement (*stage-in*) puis de rapatriement (*stage-out*) des données. Les

systèmes dépendent alors d'un système de stockage qui utilise alors généralement un système de fichiers (local).
– Les systèmes pair-à-pair : Ces systèmes de gestion de données sont usuellement caractérisés par une séparation forte des données et des méta-données et mettent en œuvre diverses techniques de recherche. Nous présenterons rapidement quelques unes de ces techniques.
– Les systèmes de fichiers : C'est le moyen le plus naturel pour accéder à ses données puisque celles-ci sont attachées au système d'exploitation. Nous présenterons rapidement les systèmes de stockage locaux. Le chapitre suivant présentera plus en détail les systèmes de fichiers plus évolués et les liens qu'ils ont avec le réseau.

3.2 Quelques mots sur les systèmes de fichiers locaux

À peu près tous les systèmes d'exploitation qui ont été créés jusqu'à présent ont intégré leur propre système de fichiers. Tous les systèmes de fichiers existants n'offrent bien sûr pas toutes les mêmes caractéristiques ni les mêmes considérations architecturales. Par exemple, certains des premiers systèmes de fichiers n'offrent pas de système d'arborescence classique car ils ne gèrent pas les hiérarchies de répertoires d'UNIX (voir le système CP/M [Tan03, partie 6.4.2]).

Dans les environnements Unix, le premier système de fichier d'UNIX [RT78] a engendré de nombreuses variations et a subi de nombreuses adaptations et optimisations pour être utilisé sur les systèmes récents et a fini par donner les systèmes tels que EXT2, FFS, UFS, ... La plupart de ces systèmes ont conservé l'utilisation d'inodes pour stocker les méta-informations ainsi que de blocs de redirection pour gérer les localisations des blocs de données. Ils ont cependant été enrichis de fonctionnalités permettant d'avoir de la journalisation, d'optimiser les accès aux gros répertoires ...)

Depuis quelques années, des systèmes à l'origine développés pour les mini-ordinateurs ont fait leur apparition sur Linux (JFS et XFS) et mettent en œuvre diverses techniques pour améliorer les performances obtenues (notamment au niveau de l'implantation avec l'utilisation de structures de données plus évoluées) et les fonctionnalités offertes par le système de fichiers : journalisation, support de réplication (XFS), de gestionnaires de volumes (XFS), ...

Dans le monde Microsoft, il y a d'abord eu les systèmes FAT qui utilisaient des tables et des listes chaînées pour stocker les fichiers. Depuis quelques années, le système NTFS est un système moderne permettant de gérer les dernières fonctionnalités offertes par le système d'exploitation : chiffrement à la volée, plusieurs flux d'octets par fichier, compactage, ...

Pour résumer cette partie courte, nous dirons que les systèmes locaux sont utilisés principalement pour stocker les données locales dans une grappe (par exemple le système et les applications). Cependant, pour que les machines d'une grappe soient utilisables, il faut souvent disposer d'un système de stockage (gros serveur par exemple) qui permet à tous les machines de stocker et partager leurs fichiers de manière transparente.

3.3 Bibliothèques d'accès à des systèmes de stockage

Le modèle retenu par de nombreuses bibliothèques est de transférer la complexité de la gestion des données en espace utilisateur.

En mettant en place des mécanismes *zero-copy*, de très bonnes performances peuvent être obtenues mais cela requiert usuellement deux choses importantes :

– un mécanisme pour éviter le traitement noyau et,
– du matériel spécialisé (donc relativement coûteux) qui met en œuvre des modes avancés de transferts (*Remote Direct Memory Access* - RDMA)

Par exemple, dans DAFS (*Direct Access File System* [MAF+02]), les applications souhaitant accéder à des données doivent être recompilées et liées avec une bibliothèque particulière qui permet de gérer les modes de transfert optimisés pour le matériel (RDMA par exemple). Dans le système ORFA[Gog03], les mécanismes d'accès optimisés s'obtiennent en utilisant un module noyau et en programmant le processeur de la carte réseau (le LANAI de la carte Myrinet). Le système READ2 [Coz03] utilise lui aussi des spécificités du matériel (Myrinet) pour offrir un accès très performant aux couches bas niveau et offre une interface simplifiée à l'utilisateur pour pouvoir gérer des accès bruts au niveau *block device* (sans système de fichiers). De telles bibliothèques très efficaces ont souvent des pré-requis matériels importants, ce qui peut aussi contribuer à en limiter l'extensibilité. Des tests de passage à l'échelle de tels systèmes encore récents et en développement n'ont pas encore été à ce jour réalisés à notre connaissance. Un autre aspect de ces approches doit aussi être pris en compte : puisque le kernel n'est plus utilisé pour gérer les fichiers, l'application doit gérer elle-même son cache de fichiers car le *buffer-cache* du kernel n'intervient plus, ce qui peut être ennuyeux ou pas selon le type d'applications.

Un autre exemple de bibliothèque d'accès (parmi de nombreux) peut être vu dans RFIO [IN2] : c'est une bibliothèque développée au CERN qui s'appuie sur une architecture client-serveur et qui offre une API C mimant l'API standard du système (`open()`, `close()`, ...) Une API C++ permet d'avoir une vision par la classe `iostream` des fichiers distants. Un démon doit être installé sur le site hébergeant les fichiers et pourra gérer les demandes de l'application.

Si l'on souhaite utiliser du matériel plus standard, des bibliothèques sont disponibles pour mettre en place une structure de systèmes de fichiers distribués. On peut par exemple penser à des bibliothèques qui définissent un stockage distribué (par exemple avec du *striping* des données). La `libpvfs` de PVFS (que nous présenterons plus en détails dans la section 4.3.3 page 64) permet par exemple aux applications conçues pour ce système d'avoir des performances accrues en ne passant pas par le VFS. Les fonctions incluses qui imitent l'API standard du système ont en fait une connaissance accrue du fonctionnement du système (répartition des données, etc...)

D'autres systèmes ont été développés mais beaucoup plus orientés pour le calcul scientifique et offrent donc des API évoluées de répartition et d'accès aux données. Parmi ceux-ci on peut par exemple citer PIOUS [MS94] (*Parallel Input/Output System*), DPFS

[SDM98] et Vesta [CF96] qui offrent une interface bi-dimensionnelle[1] pour accéder à un fichier, Galley [NK96] qui lui, offre une vision tri-dimensionnelle d'un fichier en sous-fichiers.

Par le biais de modules noyaux gérant des rappels vers les fonctions des diverses bibliothèques (à la manière des systèmes de fichiers en mode utilisateur), une intégration de ces bibliothèques peut être envisagée dans le VFS. Ainsi, elles peuvent devenir des systèmes de fichiers respectant alors plus ou moins les API POSIX, bien que cela ne soit pas souvent leur but principal.

Dans les cas décrits ci-dessus, le mode de fonctionnement envisagé était plus particulièrement un mode d'accès « à-la-volée ». Dans de nombreux cas, il est parfois nécessaire d'utiliser un mode d'accès de type « pré-chargement / traitement local / envoi ». Ce type d'accès est alors rendu possible par des systèmes de transferts.

La séparation entre bibliothèque d'accès et systèmes de transfert n'est encore pas une fois rigide car si un système de transfert permet de gérer par exemple des lectures / écritures à des *offsets* aléatoires, alors une implantation sous forme de bibliothèque d'accès comme celles que nous venons de présenter est tout à fait envisageable.

3.4 Systèmes de transfert

Comme nous l'avons indiqué précédemment, les systèmes purement de transfert permettent de mettre en place des techniques de récupération des données avant de les traiter en local (phase *stage-in*) puis éventuellement renvoyer les résultats (phase *stage-out*). Nous présentons de manière plus détaillée quelques protocoles de transferts souvent à la base des moyens des échanges de fichiers. La plupart de ces protocoles utilise le protocole TCP/IP principalement pour son omni-présence et sa gestion de la congestion.

3.4.1 HTTP

Le protocole HTTP (*Hyper Text Transfer Protocol*) est un protocole de type client-serveur conçu pour transférer les données des sites *web*. Ce protocole a commencé à être utilisé dès 1990 et est décrit dans le RFC1945 (version 1.0) [BLFF95]. La version actuellement en usage est la version 1.1, normalisée dans le RFC2616 [FGM+99].

Ce protocole est un exemple de protocole client/serveur : une connexion TCP est d'abord initiée par le client afin de pouvoir transmettre une requête, puis la réponse du serveur revient par le chemin inverse.

[1] Si le fichier représente un tableau en deux dimensions ou une matrice, le système fournit une interface pour accéder de manière simple aux données par blocs de colonnes ou blocs de lignes par exemple. Pour emprunter le vocabulaire des bases de données, on peut qualifier cela de vues du fichier.

La version 1.1 permet, entre autres, de réutiliser la connexion ouverte afin d'envoyer plusieurs requêtes de manière séquentielle sans avoir à payer à nouveau le coût d'établissement d'une nouvelle connexion. Un autre aspect intéressant est le support de l'extension *byte-range* qui permet de récupérer une partie d'un fichier (pour terminer un télé-chargement ou bien pour récupérer des morceaux d'un fichier répliqué sur plusieurs sites).

Bien évidemment ce protocole est utilisé principalement pour le téléchargement des pages *web* mais il est aussi utilisé par de nombreux systèmes pour gérer les transferts de données car, si le serveur est situé sur un port « standard », celui-ci permet de passer les systèmes pare-feu. C'est ainsi que ce protocole s'est retrouvé utilisé dans de nombreux systèmes d'échanges pair-à-pair ou bien par exemple dans dans le système GASS (voir partie 3.4.5 page 37) fourni par l'intergiciel Globus pour gérer les transferts inter-sites.

Dans le cas de transferts devant être sécures, le protocole HTTP peut aussi être utilisé au dessus d'un canal sécurisé (avec ou sans chiffrement, authentification par certificats, etc) avec les *sockets* SSL (il porte alors le nom de HTTPS).

Le protocole a été étendu par la suite afin de pouvoir offrir aux utilisateurs des fonctionnalités de partage et d'édition de documents sur machine distantes grâce aux extensions WebDAV (*Web-based Distributed Authoring and Versioning*).

3.4.2 FTP

Le protocole FTP (*File Transfer Protocol*) est un protocole développé, comme son nom l'indique, pour effectuer des transferts de fichiers, et ce, entre un client et un serveur.

FTP s'appuie sur le protocole TCP et lors d'un transfert de fichier, deux connexions sont créées. La première est une connexion de contrôle, créée à l'initiative du client vers le port du service `ftp` (usuellement port 21). La seconde peut être créée, soit à l'initiative du serveur (mode actif) - ce qui peut poser des problèmes avec certains systèmes pare-feu - soit à l'initiative du client (mode passif). Entre plusieurs transferts consécutifs de fichiers, la connexion de contrôle est conservée.

Ce protocole est décrit de manière détaillée dans le RFC 959 [PR85] et a été complété par la suite par diverses extensions pour gérer la négociation de nouvelles fonctionnalités [HE98] entre le client et le serveur ou bien pour gérer les extensions de sécurité [HL97] (par défaut, les informations [authentification, adresses, ports] circulent en clair sur le canal de contrôle, ce qui n'est pas vraiment très sûr).

FTP permet aussi de gérer des transferts à l'initiative d'un client tiers, c'est-à-dire entre deux serveurs (mode FXP) en utilisant des requêtes permettant de préciser la destination (requêtes `PORT` et `PASSIVE`). Avec les adresses de deux machines différentes, les transferts et les ports qui vont bien, le transfert peut donc être effectué. Cette opération peut devoir être autorisée explicitement sur les serveurs qui doivent accepter une connexion venant d'une machine différente de celle qui a demandé contrôle.

Cette approche peut aussi permettre de répartir la charge si un serveur « frontal » envoie les demandes de connexions à des serveurs ftp situés derrière lui ; serveurs qui

pourraient répondre alors directement aux clients. Cependant, dans ce cas-là, il faut que les clients aient un mode de sécurité particulièrement relâché en acceptant les données ne provenant pas du serveur qu'ils ont contacté, ce qui n'est pas trop recommandé d'un point de vue sécurité.

L'utilisation de deux connexions pose souvent des problèmes de sécurité sur les système pare-feu bien que cela puisse être amélioré grâce au suivi des connexions (*connection tracking* dans les noyaux Linux par exemple) mais ce suivi s'avère néanmoins plus coûteux et délicat que celui de simples connexions, comme ce qui se produit avec le HTTP.

Dans sa version de base, le FTP est considéré comme peu sécure car d'une part le canal de contrôle est dérivé du protocole `telnet` et d'autre part, le « vol de connexion » est relativement aisé si le serveur n'effectue pas de vérifications suffisamment poussées sur les sources et les destinations du transfert. De plus, cette mauvaise réputation est entachée par le fait que de nombreux trous de sécurités critiques permettant d'obtenir aisément les privilèges d'administrateur aient été découverts dans de nombreuses implantations de ce protocole.

3.4.3 GridFTP

Ce protocole de transfert pour grilles est conçu sous la forme d'extensions au protocole FTP. Les buts qu'il souhaite fournir sont décrits dans [Pro00] et les modifications du protocole sont décrites dans plusieurs documents [vH00], [ABB+01], et [Pro02].

Un des avantages est que ce protocole est défini comme nous venons de l'écrire par rapport au standard FTP, donc, à priori, il devrait pouvoir y avoir une compatibilité avec les versions « normales » de FTP. Les extensions proposées sont de plusieurs types et notamment :

- une gestion de l'authentification sécurisée en utilisant l'API GSI (*Grid Security Infrastructure*) ou bien un système *Kerberos*. Un extension est aussi prévue pour pouvoir authentifier les flux de données.
- un support pour gérer des données réparties (*stripées*) ou dupliquées afin de pouvoir accélérer les transferts (demander soit à plusieurs systèmes d'envoyer les morceaux du fichier ou bien demander des morceaux de fichiers à plusieurs systèmes afin d'alléger les entrées/sorties sur ces serveurs). Le comportement défini à suivre est implanté grâce à l'ajout de commandes `FTP` : `SPAS` (*Striped Passive*), `SPOR` (*Striped Data Port*), `ERET` (*Extensive Retrieve*), `ESTO` (*Extensive Store*).
- il y a de plus des extensions prévues pour optimiser, entre autres, les modes de transferts des couches réseaux utilisées (par exemple, un mode d'auto-négociation des fenêtres TCP afin d'optimiser les transferts). Ceci est réalisé grâce aux commandes SBUF (*Set Buffer Size*) et ABUF (*Auto-Negotiation Modes*).
- les flux parallèles peuvent aussi être utilisés d'une manière similaire à celle qui est utilisée par les accélérateurs de téléchargement : en multipliant le nombre de

connexions TCP ouvertes entre la source et la destination, les performances sont souvent meilleures mais ceci a pour contre-partie de saturer les serveurs et d'éprouver les politiques de contrôle de la congestion utilisées dans le protocole TCP.

Une implantation-prototype de ce protocole a été développée sous forme de *patches* pour le serveur `wu-ftpd` et le client `ncftp`. À terme, plusieurs services devraient pouvoir s'appuyer sur ce noyau GridFTP tels que des clients et des serveurs utilisant des services tels que GASS ou HPSS.

3.4.4 BBFTP (Babar FTP)

Ce système client/serveur est développé par l'IN2P3 afin de permettre d'échanger le plus rapidement possible des données entre deux sites.

Pour ce faire, un canal de contrôle sécurisé (SSL) permet de gérer plusieurs connexions TCP de données en parallèle. Les données ne sont cependant pas cryptées pour des raisons de performance. De même que pour le FTP, le client initie une connexion de contrôle, puis à travers celle-ci demande l'ouverture de plusieurs flux TCP dans des *sockets* ouverts auparavant en réception.

Le mode de transfert géré est un mode point à point entre deux machines. L'utilisation de la bande passante est alors maximisée : si un des flux TCP subit une perte de paquets et donc est amené à ralentir son débit, un des autres flux de l'application récupérera alors rapidement la bande passante libérée en attendant qu'un mode stable se rétablisse. Globalement, la vitesse de transfert pourra donc être plus élevée qu'avec un seul flux (et ce au détriment des flux d'autres applications qui pouvaient transiter).

3.4.5 GASS (*Global Access to Secondary Storage*)

Cet acronyme[BFK+99] désigne un composant intégré à l'infrastructure Globus pour gérer les accès aux fichiers. Son rôle est de fournir un cache de fichiers qui puisse gérer certains des modes d'accès typiques aux fichiers utilisés dans les applications scientifiques.

À la manière des systèmes de fichiers distribués CODA et AFS que nous verrons par la suite en section 4.3.1, les fichiers sont récupérés lorsqu'une application souhaite ouvrir le fichier. Elle manipule alors une copie locale de cet objet puis lorsque le fichier est libéré, la copie peut être renvoyée vers le serveur de stockage. Ce mode est donc tout particulièrement indiqué pour les types d'accès de type *stage-in/stage-out*.

Un autre mode permettant d'offrir une fonctionnalité `APPEND` est aussi disponible et permet donc de gérer les fichiers de type *log*.

Au niveau des serveurs GASS, de la glue logicielle permet de s'interfacer sur plusieurs sous-systèmes de stockage et donc de gérer de cette manière plusieurs systèmes de stockage (systèmes de fichiers classiques ou bibliothèques d'accès à des données, par exemple).

Les fichiers sont repérés par une adresse de type URI du style `https://host:port/file` et sont transférés dans leur intégralité dans un cache local avant que l'application ne puisse y accéder. Pour le transfert, les protocoles tels que `FTP`, `HTTPS` ou `x-gass` (le protocole de l'implantation peuvent être utilisés). Actuellement, le mode de transfert retenu par défaut est un mode de transfert par HTTPS.

3.4.6 IBP (*Internet Backplane Protocol*)

IBP[ABM+02][BBMP02] est une couche d'intergiciels destinée à gérer et à utiliser des systèmes de stockage distants et vise à mettre en place un support de « Réseau Logistique » (*Logistical Networking*). Ce dernier concept consiste principalement à importer dans le domaine informatique des techniques telles que l'utilisation d'entrepôts (gros sites de stockage), de dépôts (sites locaux à de nombreux endroits) et de canaux de distributions (les « tuyaux » réseaux). Ainsi, ces moyens permettent d'obtenir système gérant les transferts et réplications de données entre dépôts afin de les rapprocher du ou des sites qui doivent les utiliser. Ce concept de « tampons » proches est aussi utilisé dans certains systèmes de distribution de contenu tels que le réseau Akamai [MIT02] ou bien sur les gros sites web qui redirigent l'utilisateur sur un miroir plus proche.

Ce service est fourni à l'utilisateur sous la forme d'une API permettant d'accéder à plusieurs sources de stockage distantes (dépôts et ressources Internet « classiques ») d'une manière uniformisée. Cette interface a pour but d'offrir un standard, inter-opérable et extensible, aussi l'API doit-elle être limitée pour pouvoir supporter aisément plusieurs systèmes de stockage :

- allocation/location de place (*allocate*) : le client demande de l'espace pour une durée variable et en cas de succès, trois clés lui sont renvoyées et chacune aura un rôle et un pouvoir spécifiques (lecture, écriture et administration)
- écriture (*store*) : rajouter des données à la fin de son espace alloué
- lecture (*load*) : lire des données depuis son espace alloué
- transfert entre deux sites (*copy*) : comme pour le transfert par FXP dans le protocole FTP
- transfert d'un site vers plusieurs (*mcopy*) : grâce à des déplaceurs de données (*Data Mover*), plusieurs modes de diffusion peuvent être envisagés (techniques *multicast* par exemple)
- gestion de l'allocation (*manage*) : prolongation ou diminution de la durée de l'allocation,

Afin d'offrir une utilisation plus aisée, plusieurs services gravitent autour de l'infrastructure IBP :

- L-Bone : ceci désigne un service d'indexation et de monitoring par NWS[Wol98] des dépôts IBP publics
- ExNodes : cette bibliothèque permet d'abstraire l'agrégation de plusieurs allocations IBP

– LoRS (*Logical Runtime System*) permet d'automatiser certaines opérations relatives à l'utilisation de IBP (primitives pour gérer les réplicats) et fournit également des moyens de chiffrer et calculer des sommes de contrôle des données lors du transfert

Plusieurs applications requérant de gros besoins en stockage et bande passantes sont actuellement expérimentées pour la distribution d'images ISO, la mise en place de *streaming* multimédias, ...

3.4.7 Conclusion

Nous venons donc de présenter quelques protocoles et systèmes usuels permettant de gérer les transferts de fichiers. Ceux-ci seront plutôt utilisés dans le cadre de scénarios d'utilisation *stage-in / stage-out*, autrement dit : les données sont récupérées sur un stockage local, traitées puis renvoyés sur le stockage source.

Un autre type de systèmes de transfert est constitué par les systèmes pair-à-pair : cette thématique commence à rentrer dans le monde des systèmes de stockage, aussi allons-nous présenter dans les grandes lignes quelques techniques intéressantes utilisées par ces systèmes.

3.5 Systèmes pair-à-pair

Ces systèmes qui constituent souvent le cauchemar des fournisseurs d'accès Internet, des sociétés d'éditions et des administrateurs pour diverses raisons, sont apparus comme des produits de l'industrie logicielle (Napster a été l'un des premiers à être fortement médiatisé, ce qui a d'ailleurs plus ou moins causé sa perte...). Ils ont, après coup, commencé à intéresser la communauté scientifique qui a étudié et développé de nouveaux modèles d'échanges.

Une classification des différentes familles et aussi leurs particularités, notamment en ce qui concerne le placement et la recherche des données, sont disponibles dans [MKL+02].

De manière générale, on trouve des problèmes similaires à ceux rencontrés dans les systèmes de fichiers distribués, principalement issus de la non-centralisation des données. On peut évoquer les points suivants :

– publier les mises à jour : en général cet aspect est peu présent dans les systèmes distribués car les données une fois mises en ligne peuvent être considérées comme des objets non mutables - on notera cependant que certains systèmes s'intéressent à ce problème, notamment dans le cas de partage de documents collaboratifs (*groupwares*) ou bien prototypes de recherche (voir par exemple [RCN03])
– initialiser le système (*boot-strapping)* : « si je ne connais personne comment récupérer un premier pair qui me permettra de rentrer dans la communauté ? »
– les problèmes de partage des données

– confidentialité : comment être sur que les données ne sont pas corrompues ou bien qu'elles n'ont pas été modifiées lors des transferts
– les problèmes de résilience : c'est une version extrême des problèmes de tolérance aux pannes appliquée aux systèmes de fichiers distribués car, les machines reliées en P2P ont en général de mauvaises propriétés (disparitions, déconnexions, ...) de par la présence d'utilisateurs interactifs.

Le problème du *boot-strapping* est en général résolu par le fait que les logiciels de partage embarquent une liste de points fixes qu'ils sauront contacter quand ils s'initialiseront pour avoir de nouveaux points de contacts. Les points fixes sont ensuite souvent limités à jouer seulement le rôle de courtier (*broker*), ce qui permet souvent d'alléger leur charge.

La plupart des systèmes de partage de fichiers implantent aussi des techniques permettant de gérer le téléchargement de fichiers par morceaux provenant de plusieurs pairs (*E-Donkey*, *KaZaA*, ...)

Par exemple dans *BitTorrent*[Coh] l'équivalent des méta-données est un fichier contenant les informations pour récupérer le fichier ainsi que l'adresse d'un *tracker* qui met en relation les pairs. D'autres systèmes utilisent des « super-pairs » (*KaZaA*, *Gnutella2*), afin d'éviter que les machines trop lentes ne handicapent le réseau, ceux-ci jouent alors le rôle de concentrateur et peuvent être récompensés par l'octroi de bonus par le système (par exemple des crédits pour effectuer des recherches dans *MojoNation*).

Une autre notion aussi à prendre en compte pour la stabilité globale des stockages de type pair-à-pair est la robustesse du système. En effet, ces systèmes étant souvent ouverts sur le réseau, la question qui doit se poser est « est-ce que le protocole du système est suffisamment robuste pour accepter des clients ne respectant pas les recommandations du protocole ? ». Si la présence de quelques clients officieux risque d'affecter le fonctionnement global du système et de faire une sorte de déni de services, la qualité d'accès aux données va alors bien évidemment grandement en souffrir.

Certains systèmes de fichiers commencent à utiliser des techniques de stockage très inspirées de ces modèles pair-à-pair et on peut notamment évoquer le système CFS (*Cooperative File System*) construit au dessus de SFS (voir section 4.3.4 page 66). Ce système a de nombreuses similarités avec les systèmes pair-à-pair dont il utilise certains des algorithmes de recherche (CHORD [SMLN$^+$02]).

Nous venons donc de présenter des moyens d'accéder aux données par le biais d'un système pair-à-pair notamment car plusieurs des techniques utilisées au sein de ces environnements peuvent à notre avis s'avérer intéressantes dans le cadre d'une utilisation sur grilles de grappes : l'utilisation de plusieurs sources de données ainsi que la recherche des fichiers (méta-données) est en effet un problème intéressant dans ce cas. De plus, par exemple, lors du démarrage d'une application parallèle, le fait que certains nœuds ayant déjà le fichier fournissent des données aux autres peut s'avérer intéressant afin de réaliser un démarrage plus rapide du programme.

3.6 Conclusion

Nous venons donc de présenter divers moyens d'accéder à des données à savoir par bibliothèques (en accès « à la volée »), par transferts explicites (mode *stage-in/stage-out*) et par modèles pair-à-pair (l'un ou l'autre type d'accès selon le système). Le tableau 3.1 effectue un récapitulatif des moyens d'accès aux données.

Le chapitre suivant va concerner les systèmes de fichiers, c'est-à-dire les systèmes qui permettent à l'utilisateur d'utiliser de manière standard les données, sans avoir besoin d'utiliser des bibliothèques spécifiques supplémentaires, mais simplement les appels systèmes classiques.

Nom	Environnement				Fonctionnalités			Pérennité	
	Intrusivité	Pré-requis	Linux	API	Données	Méta-données	Cohérence	État	Licence
FS local	+++	-	*	VFS	0	0	+++	++	?
Bibliothèque accès distants	+	0→+++	-→*	sp.	0→++	0→++	0→+++	-→++	?
Bibliothèques DFS distribués	+	0→+++	-→*	sp.	0→++	0→++	0→+++	-→++	?
Transfert : HTTP	0→+	0(+)	*	sp.	0	0	+	++	S/O
Transfert : FTP/BBFTP	0→+	0(+)	*	sp.	0	0	0	++	S/O
Transfert : GridFTP	0→+	0(+)	*	sp.	0→++	0	0	++	S/O
Transfert : GASS	0→+	0(+)	*	sp.	0	0	0→+	+	O
Transfert : IBP	0→+	0(+)	*	sp.	0	0	0	++	O

TAB. 3.1 – Récapitulatif des systèmes d'accès aux données
Les critères ont été définis dans le tableau 2.1 page 30.

Chapitre 4

Systèmes de fichiers

Grâce au système proposé par le VFS le support pour plusieurs systèmes de fichiers peut être offert et ce de manière quasi-transparente pour l'utilisateur : si ce dernier ne se renseigne pas plus que cela pour savoir sur quel type de partition est situé le fichier auquel il accède, il peut très bien ne pas savoir que l'accès peut être fait à plus bas niveau par le biais d'un réseau ou bien en mémoire, etc…

C'est ainsi que l'application peut accéder à n'importe quel fichier de la VFS par les primitives standards (`open()` / `read()` / `write()` / `close()`) sans se soucier de savoir ce qui se passe en dessous, dans le système.

Nous avons déjà évoqué rapidement le cas des systèmes de fichiers locaux dans le chapitre précédent (cf. 3.2 page 32), aussi allons-nous maintenant nous intéresser plus particulièrement aux systèmes de fichiers utilisant des réseaux, que ce soient des réseaux classiques (Ethernet) ou bien des réseaux dédiés de stockage (systèmes SAN).

Ce chapitre va donc présenter plusieurs systèmes de fichiers se servant du réseau pour fournir un service de gestion de fichiers aux diverses machines.

4.1 Systèmes de fichiers réseau/d'export

Lorsqu'une application doit accéder à des données stockées sur un système de fichiers local par le biais du réseau, une solution consiste à installer un serveur sur la machine contenant les données pour pouvoir autoriser l'accès par le réseau. De tels systèmes ont souvent une architecture clients-serveur : le terme utilisé pour caractériser cela est alors « export réseau » ou « volume réseau ».

Nous allons présenter quelques moyens usuels utilisés dans le monde Unix (NFS) ou Windows (CIFS). Divers autres moyens peuvent être utilisés pour exporter des données et notamment les solutions de travail collaboratif (*groupware*) mais celles-ci, d'une part, se situent à un niveau logiciel beaucoup plus élevé et ne correspondent pas vraiment à la problématique à laquelle nous nous intéressons (systèmes pour grappes Beowulf) et. d'autre part, n'entrent pas souvent dans la catégorie systèmes de fichiers.

4.1.1 NFS

Le protocole NFS (*Network File System*) est disponible actuellement dans 3 versions qui portent les numéros 2,3 et 4 - la version 1 étant restée un prototype interne chez son concepteur, Sun Microsystems. Ce protocole utilise un encodage XDR des données et le système de RPC développé originellement par Sun.

4.1.1.1 NFS Version 2

C'est un protocole client-serveur synchrone sans état, c'est-à-dire que le client envoie une requête qui contient tout ce qui sera nécessaire à sa résolution et attend la réponse. Il est normalisé dans le RFC1094 [Sun89] et est disponible pour les réseaux à base de TCP et d'UDP, cependant la nature « sans-état » du protocole fait que souvent le protocole UDP est utilisé.

Le fait d'être un protocole sans état a plusieurs avantages notamment en cas de *timeout* ou de panne d'un des composants : si une requête est perdue (sur le réseau ou si le serveur est en panne par exemple), il suffit de la ré-émettre jusqu'à obtention d'une réponse. Certaines requêtes idem-potentes peuvent être envoyées plusieurs fois sans problème, par contre d'autres peuvent poser certains problèmes.

En effet, on peut, pour illustrer cela, prendre le cas où un message d'acquittement de l'effacement d'un fichier est perdu : le client va redemander l'effacement de ce fichier et aura une erreur. Pour éviter ces problèmes, les serveurs s'autorisent souvent une entorse au fait d'être sans-état en maintenant un cache des couples requête-réponse récemment renvoyés.

Dans les échanges avec le serveur, les fichiers et les dossiers sont représentés dans ces transactions par des *handles* NFS (des identifiants) qui sont une suite d'octets opaque, ce qui signifie que ces octets n'ont un sens que pour le serveur.

Afin d'obtenir ces références, le client utilise la procédure de `LOOKUP` qui permet de convertir un nom de fichier en *handle*. Cette procédure prend un *handle* de répertoire et un nom (de fichier, de répertoire, etc) puis renvoie le *handle* de l'objet demandé ou une erreur, s'il n'existe pas. Puisqu'il faut un *handle* initial du répertoire exporté pour démarrer le système, un protocole ancillaire a été ajouté pour gérer ce besoin et est implanté à l'aide d'un démon `mountd`, qui peut calculer des *handles* NFS pour les répertoires exportés.

Le mode de fonctionnement de ce serveur de blocs peut être illustré avec le mini-scénario suivant qui se produit lors d'une lecture de données :

1. Requête `LOOKUP` : `handle_dir`, « `nomfichier` »
2. Réponse `LOOKUP` : `handle_nomfichier`
3. Requête `READ` : `handle_nomfichier`, *`offset`*, `taille`
4. Réponse `READ` : données

Afin de conserver des bonnes performances, les données et les attributs de fichiers sont usuellement cachés sur le client pendant une certaine durée car le système utilise une cohérence temporelle. Par exemple, un client 1 peut créer un fichier mais un autre client 2 ne va pas voir la mise à jour effectuée par le client, car le client 2 a déjà caché le contenu du répertoire avant la mise à jour. Un autre exemple similaire existe avec des données cachées : le client 2 lit des données et les cache pendant 3 secondes, le client 1 met les données à jour sur le serveur, si le client 2 relit les données avant leur expirations, l'application verra les données cachées et pas les plus récentes.

Certains travaux (NQ-NFS [Mac94][MBKQ96]) sur NFS ont souhaité améliorer ce modèle en mettant en place un mécanisme utilisant des locations de courte durée et des *callbacks* [1] afin d'offrir une meilleure gestion de la cohérence des caches.

La synchronisation des accès peut être effectuée à l'aide d'un autre protocole nommé NLM (*Network Lock Manager*) en version 1 ou 3. Les fonctionnalités de ce protocole de verrouillage sont alors fournies par le biais d'un service RPC supplémentaire.

4.1.1.2 NFS Version 3

Cette version[CPS95] constitue une évolution de la version 2 qui en corrige certaines limitations, principalement pour en améliorer les performances. Il y a relativement peu de modifications majeures mais on peut par exemple évoquer :

- la taille des *handles* NFS a été augmentée : d'un bloc fixe de 32 octets dans la version 2, NFS3 est passé à des *handles* de taille variable d'au maximum 64 octets ;
- le support des « gros fichiers » (dont la taille ne tient pas sur 32 bits) ;
- le support des écritures asynchrones avec un *commit* synchronisant : le client envoie une série d'écritures pour laquelle il reçoit un acquittement et s'il veut être sûr que ses données ont été stockées sur support stable, il envoie alors une requête `COMMIT` ; une fois l'acquittement reçu, il pourra ainsi en être sûr.
- l'ajout d'opérations composées (par exemple obtenir la liste des fichiers d'un répertoire et leurs informations en un seul appel)

De même que dans la version 2, des serveurs RPC annexes sont encore nécessaires pour initialiser le système et pour en gérer les verrous (NLM version 4).

4.1.1.3 NFS Version 4

Cette version[PSB+01][CRT+00] consiste en une refonte majeure des versions précédentes : d'un protocole sans état commun aux versions 2 et 3, cette version introduit une sémantique de « `open` » et de « `close` » et devient donc avec état.

Cette version souhaite résoudre les problèmes qu'avaient NFS à plusieurs niveaux :

[1] Le client devient alors en quelque sorte aussi serveur, car le serveur prend l'initiative de recontacter le client.

– des performances améliorées lors d'une utilisation sur Internet : ceci n'était pas vraiment le cas des versions précédentes qui, par manque de moyen sûr d'authentification et par leur architecture souvent dépendante de la latence du réseau (surtout en UDP), ne permettaient pas un passage performant sur un tel réseau.
– une gestion améliorée de la sécurité des données et de l'authentification des utilisateurs : cette limitation des versions précédentes est traitée avec l'utilisation des `RPCSEC_GSS` et d'une infrastructure de type *Kerberos*. Un nouveau type de message permet aussi au client de s'informer et de négocier la politique de sécurité du serveur, alors qu'une session peut déjà être établie.
– inter-opérabilité améliorée entre plateformes : le modèle offert par NFS jusqu'alors était, malgré les efforts faits, très dépendant du modèle de fichiers des Unix notamment pour la gestion des permissions (par exemple avec listes de contrôle d'accès).
– permettre des extensions futures (par exemple pour les permissions...)

Ce protocole introduit aussi les requêtes de types `COMPOUND` qui correspondent en fait à une agrégation de sous-requêtes au sein d'une seule requête RPC. À la première erreur, l'exécution est arrêtée et le message d'erreur est retourné au client. Pour pouvoir agréger les sous-requêtes de manière efficace un système de mémoire pour la requête est disponible sur le serveur ce qui permet par exemple de grouper un `LOOKUP` avec un `OPEN` et un `READ`.

Un autre aspect intéressant est la gestion des attributs qui permet par exemple de gérer la migration ou la réplication des données (en mode lecture seule) : si un fichier est déplacé, un attribut permet d'indiquer où se trouvent les nouveaux fichiers. De plus une meilleure gestion des caches peut être offerte par le biais de locations (comme dans NQ-NFS) ou délégations (le client gère les verrous lui même sur le fichier s'il en est le seul utilisateur).

Un des facteurs limitants de cette nouvelle version est que à ce jour, les implantations existantes sont relativement jeunes et, de plus, il apparaît comme un manque de solutions industrielles proposant ce système.

4.1.1.4 Conclusions sur NFS

Les versions 2 et 3 de NFS sont un standard dans le monde UNIX malgré les défauts qu'elles ont ; défauts tels qu'entre autres, la cohérence faible des caches (cohérence temporelle). Cependant, son omniprésence peut s'expliquer par plusieurs raisons : c'est un standard relativement ancien, facile à configurer et de plus, il est supporté par tous, sinon une très grande proportion, des systèmes Unix.

La version 4 ne dispose pas à l'heure actuelle de base d'utilisateurs vraiment établie et est encore relativement peu disponible, probablement car elle a subi, principalement, de profonds remaniements par rapport aux versions 2 et 3.

4.1.2 CIFS (*Common Internet File System*)

Ce protocole fermé (bien qu'une tentative de normalisation sous forme de RFC ait été effectuée) à l'origine est devenu un protocole de fait car il est disponible sur toutes les machines utilisant les systèmes de Microsoft. Il s'est d'abord appelé SMB (*Server Message Block*) puis s'est vu renommé récemment CIFS. De par son omniprésence, les vendeurs de solutions de stockage intégrées offrent souvent des accès au stockage par NFS et CIFS.

Une implantation en Open Source (SAMBA) a pu être réalisée par *reverse-engineer*ing du protocole et permet aux Unix de servir de serveurs CIFS. Bien évidemment, chaque nouvelle version majeure de *Windows* ajoute son lot de modifications et d'incompatibilités avec cette application mais cela ne l'empêche pas d'être le système d'échange de fichiers de choix pour les environnement Microsoft.

De même que NFS, ce protocole définit un moyen d'exporter des systèmes de fichiers déjà existants mais il permet en plus d'exporter d'autres services et notamment les services d'impressions. En revanche, contrairement à NFS, ce protocole est orienté connexion et doit donc gérer des sessions avec les états associés. C'est ainsi qu'un accès à une ressource partagé se déroule usuellement en trois phases :

1. négociation du protocole à utiliser dans la suite de l'échange
2. établissement d'une session : cette phase regroupe l'authentification du client et l'envoi en cas de succès d'un identifiant de session
3. établissement de la connexion à l'arbre (*tree connection*) : le client s'il a les crédits nécessaires peut accéder à la ressource et reçoit un TID (*Tree connection ID*) qui sera utilisé dans toutes les requêtes suivantes d'accès aux fichiers

Les étapes 2 et 3 peuvent être plus ou moins altérées en fonction du modèle de sécurité retenu par les différents systèmes.

À partir de *Windows 2000 Server*, Microsoft a mis en place un système permettant d'agréger les différents volumes au sein d'une arborescence unifiée au sein de DFS (*Distributed File System*[Mic99]). Ainsi, ce système permet de mettre en œuvre des sortes d'alias réseau (*junction point*) : ceux-ci permettent d'indiquer au système l'endroit où aller chercher la ressource devant être récupérée.

Diverses extensions permettant de gérer de manière avancée des caches sur les clients ont aussi été ajoutées (*Client Side Caching*), ce qui peut permettre à certains clients de gérer les déconnexions ainsi qu'un cache local en « dur » (si la fonctionnalité est disponible sur le serveur).

4.1.3 MFS (*Mosix File System)*

MFS (*Mosix File System*) est le système développé par le projet MOSIX qui lui même est développé à l'Université de Jérusalem.

Pour mémoire, le système Mosix est une modification du noyau de l'OS qui permet d'agréger plusieurs nœuds d'un *cluster* de manière transparente en offrant une répartition de charge entre les différentes machines par le biais de techniques avancées de migration de processus[2] et de répartition de charge.

MFS permet à une grappe Mosix de partager aisément des fichiers entre plusieurs nœuds. Il offre en effet à cet usage une unicité de nommage à travers un `/mfs`. Les fichiers sont alors accessibles par tous les nœuds par un chemin du type :
`/mfs/<nom_du_noeud>/chemin/fichier`.

L'approche est similaire à ce que l'on pourrait obtenir avec des configurations utilisant NFS et `automount` mais MFS offre des garanties supplémentaires telles que la cohérence des caches, des *timestamps* et des liens.

Ce système est aussi optimisé pour pouvoir fonctionner en mode DFSA (*Direct File System Access*) qui permet d'améliorer les performances des processus migrés accédant à des données MFS. De manière extrêmement simplifiée, une espèce de morceau du processus reste sur son nœud de lancement et va agir comme mandataire (pour gérer les appels systèmes) entre le nœud originel et le nœud d'exécution. DFSA permet aux processus des systèmes Mosix d'accéder aux fichiers MFS sans avoir à repasser par le nœud initial du processus, si celui a été amené à migrer (en quelque sorte en court-circuitant les appels systèmes effectués).

4.1.4 Conclusion sur les systèmes d'exports

Ces systèmes prennent un système de fichiers déjà existant (la plupart du temps local) et le ré-exportent vers d'autres machines. Étant intégrés nativement dans de nombreux systèmes (NFS est omni-présent dans le monde Unix, CIFS dans le monde *Windows*) une utilisation de ces protocoles permet d'avoir une réutilisation du client natif et donc d'éviter de surcharger les clients par de nouveaux composants systèmes.

Un des désavantages de ces systèmes est leur nature centralisée ; pour obtenir des performances dans le cadre d'un serveur de fichier, il donc est souvent nécessaire de recourir à des investissements matériels.

En ce qui concerne NFS, plusieurs modèles ont été développés pour optimiser son fonctionnement et ses performances. Nous évoquerons plusieurs de ces travaux dans la section 5.2 page 72.

4.2 Systèmes de fichiers distribués à disques partagés

Ces systèmes requièrent la plupart du temps du matériel dédié (baies SAN) et sont caractérisés par le fait que plusieurs machines puissent accéder en parallèle à un même

[2] Un « morceau de processus » reste sur le nœud où le processus a été lancé afin de pouvoir exécuter les appels systèmes pour le compte de l'unité d'exécution distante.

périphérique physique (comme si le disque dur était branché sur deux machines simultanément). Bien évidemment, cette approche requiert une synchronisation des machines sous peine de voir très rapidement des corruptions des systèmes à cause des accès concurrents à une même zone. Les problèmes que doivent donc résoudre ces systèmes sont donc, par nature, très proches de ceux présents dans la programmation *multi-threadée*, ou bien de ceux qui sont observés dans les systèmes de mémoires distribuées partagées.

4.2.1 CXFS (*client-server cluster technology*)

CXFS[Sil] est une solution proposée par SGI et s'appuie sur l'architecture de stockage mise en place dans le cadre de leur système de fichiers local journalisé XFS. L'architecture retenue est une architecture de type client-serveur avec baies de disques partagées (SAN).

Les disques stockent les données et celles-ci sont lues ou écrites par le biais de canaux directs (disques partagés). En revanche, la gestion des méta-données est faite sur un mode clients-serveur plus classique. Ainsi, un serveur dédié est censé gérer l'intégralité des méta-données pour un système de fichiers. Le stockage des méta-données utilise le système de stockage local XFS ; les communications entre clients et serveurs de méta-données ont lieu par le biais de canaux TCP sur un réseau différent de celui du SAN. Le gain de performances de cette architecture vient du fait que seules les méta-données vont circuler sur ce réseau et ne pas le surcharger car leur taille est petite par rapport aux opérations lourdes en E/S que sont les lectures et écritures de données.

Lorsqu'un fichier est ouvert sur un client, ce dernier va récupérer la méta-donnée associée et les accès sur ce fichier ouvert ne vont pas entraîner de nouvelles récupérations de ces méta-informations. Afin de gérer les synchronisations sur les méta-données, celles-ci sont séparées en diverses zones d'autorité telles que les dates d'accès, la taille, la localisation des données… et chacune de ces sous-méta-données dispose d'un système de jeton qui, lorsqu'il est accordé, donne le droit au client de cacher les informations associées. La perte du jeton signifie que le client doit cesser d'utiliser les informations cachées et qu'il doit récupérer des informations fraîches sur le serveur. Ainsi, avec une telle approche, les seules mises à jour à transmettre au serveur de méta-données se produiront quand le client fermera son fichier ou bien lorsqu'il voudra modifier le fichier (demande d'allocation de place par exemple).

En ce qui concerne la tolérance aux pannes du système, un système de surveillance par *heartbeat* est disponible afin de détecter rapidement les défaillances qui peuvent avoir des conséquences désastreuses dans la mesure où plusieurs clients peuvent accéder à des mêmes données en parallèle. Un niveau de redondance peut être assuré par une réplication des méta-données à la volée et celles-ci bénéficieront de la journalisation offerte par le système local XFS.

4.2.2 Famille GFS (Sistina Software et OpenGFS)

Ce système a commencé à être développé en 1996 et visait à l'origine à tirer le meilleur parti de disques SCSI reliés entre eux par de la fibre optique (technologies avec protocole *Fibre Channel*). En août 2001, *Sistina Software* a décidé de procéder à un changement de licence de leur produit GFS, et peu de temps après, le projet OpenGFS a « *forké* ».

De nombreuses publications présentent le système et son évolution depuis cette date.

La version 1 [SRO96] a introduit le concept de verrous offerts par le matériel. En effet, les concepteurs de GFS ont participé à l'élaboration d'extensions au jeu d'instructions SCSI pour que les disques puissent gérer des verrous. Ainsi, chaque périphérique dispose d'un certains nombre de verrous (1024 verrous, par exemple) qui peuvent être activés par une instruction SCSI : ce sont des *dlocks* (pour *device lock*) qui fonctionnent selon un modèle *test-and-set* et *clear*. Si ceux-ci ne sont pas disponibles un verrou à gros-grain (c'est-à-dire sur tout le disque !) peut être utilisé avec une autre série de commandes SCSI qui ne sont qu'optionnelles (*Reserve* et *Release on Extents*). Il ne faut cependant pas oublier que, même avec des *dlocks*, les disques n'ont pas la notion de la donnée que le verrou protège et la charge incombe au système de maintenir la correspondance verrou-donnée protégée.

L'organisation des données est gérée de manière un peu plus classique : un super-bloc (non-distribué) contient les références sur des groupes de ressources qui peuvent être *stripés* sur plusieurs disques (ce sont des espèces de partitions logiques). Chaque groupe de ressources est constitué d'un tableau d'allocation des inodes et des blocs de données ainsi que, bien sûr, des inodes et des données, sans oublier d'un certain nombre de *dlocks* (nombre dépendant du matériel).

Alors que la version 2[SEP+97] a principalement été le développement d'un implantation pour IRIX, la version 3 [PBB+99] a, en plus d'être une adaptation pour Linux, apporté plusieurs optimisations au système notamment en changeant les implantations de plusieurs de ses éléments (gestion des répertoires, mise en place de verrouillage récursif, utilisation du *buffer-cache*, ...)

La version 4 [SPCSLCS00][PBB+00] apporte plusieurs améliorations notamment un support de la négociation entre les nœuds pour la gestions des *glocks* (*global locks* = verrous globaux) qui peuvent être (ou pas) des *dlocks*, un gestionnaire de verrous s'occupant de convertir les *glocks* en structure adéquate - un serveur centralisé de verrous peut même faire l'affaire, bien que les performances risquent de ne pas être très bonnes. Cette version ajoute, de plus, un support de la journalisation afin d'améliorer la reprise après panne.

Dans toutes les versions, les méta-données ne sont pas stockées sur un seul disque mais réparties sur plusieurs disques et rassemblées en groupes (d'une manière un peu similaire à ce qui se fait par exemple dans le système local EXT2 avec les groupes de blocs). Un super-bloc reste cependant non-distribué et contient des renseignements sur les données distribuées au sein des nœuds (notamment la localisation des groupes ainsi que diverses informations telles que la taille des blocs utilisés).

Un mode de compatibilité a été développé grâce à une implantation logicielle mais les performances s'avèrent bien évidemment moins remarquables que celles offertes par du matériel. Des expérimentations avec GFS sont actuellement menées au sein de l'équipe SARDES afin de voir si l'utilisation de réseaux rapides (par exemple de type SCI ou Myrinet) peuvent améliorer ce fonctionnement (sans passer par l'utilisation d'une coûteuse baie SAN), notamment au sein du projet Proboscis[HL02].

4.2.3 Frangipani/Petal

Frangipani[TML97] constitue la partie système de fichier et s'appuie sur Petal[LT96] pour la gestion des volumes (agrégation des disques et distribution des données stockées).

D'une manière un peu similaire à ce qui est développé dans GFS, le système de fichiers proposé (Frangipani) utilise un volume distribué sur plusieurs disques (Petal).

La couche Petal gère un groupe de serveurs avec disques qui coopèrent afin d'offrir aux couches supérieures une vision unifiée d'un (ou de plusieurs) périphérique de stockage. Les clients de cette couche communiquent par le biais de RPC avec les serveurs : le problème principal que ces derniers doivent résoudre correspond à assurer la transformation des demandes du client du type `<disque-virtuel:offset>` en demandes compréhensibles par les serveurs, c'est-à-dire de type`<serveurphysique:disquephysique:offsetphysique>`. Lors de la conception de cette couche, les développeurs du modèle ont souhaité que les clients aient le moins d'informations à gérer et donc, ces derniers ne disposent que d'une indication (*hint*) éventuellement erronée sur le serveur apte à gérer leur requête. En cas de mauvaise information, le client reçoit une erreur et il va alors devoir mettre à jour ses informations puis ré-émettre la requête.

Les structures permettant de résoudre la translation consistent en un ensemble de 3 tables organisées d'une manière hiérarchique :

1. répertoire de disque virtuel (VDir = *Virtual Disk Directory*) qui est répliqué sur tous les nœuds et assure donc la conversion disque-virtuel => Gmap (décrit ci-dessous)
2. carte globale (Gmap = *Global Map*) : ce second niveau (lui aussi répliqué sur tous les serveurs) permet de trouver le serveur (et son disque physique) responsable de l'*offset* demandé
3. le troisième niveau (propre à chaque serveur) permet de terminer la conversion de l'*offset* demandé en offset physique sur le disque

Des fonctionnalités de prise d'image (*snapshots*) sont aussi prévues et supportées par le biais d'un ajout d'une version des données.

Sur le périphérique virtuel Petal, un système de fichiers classique peut être installé mais, afin de tirer le meilleur parti de cette nature distribuée, une couche dédiée a été développée en tirant partie de cette nature distribuée et porte le nom de Frangipani.

Cette dernière gère uniquement le système de fichiers (fichiers, répertoires) mais reste cependant très dépendante d'une fonctionnalité fournie par la couche de stockage distribuée : en effet, le disque virtuel Petal est vu comme un disque de 2^{64} octets mais seuls les blocs effectivement utilisés bénéficient d'une allocation sur support stable (comme dans le cas des fichiers creux[3]).

L'espace de stockage est découpé en grosses zones servant à gérer les *logs*, les *bitmaps*, les inodes et les blocs de données (blocs de 4KB et gros blocs de 1TB). Chacune des zones de *logs* et de *bitmaps* est alors re-découpée en zones d'utilisation exclusives pour un des serveurs (ce qui permet de simplifier la gestion des verrous). La correspondance bitmaps d'allocation-inode est fixée, ce qui revient aussi à re-découper l'espace des inodes en « zones d'autorité ».

Frangipani utilise un service de verrouillage sous la forme de locations à renouveler avec des verrous de type lecteurs-écrivain. Si ce service souhaite passer un verrou du niveau écriture au niveau lecture, les serveurs doivent vider leur caches sur disque avant d'acquitter la demande.

La gestion de la tolérance aux pannes est assurée par un système de *logs* sur les serveurs ainsi que des mécanismes de détection par *heartbeat*. Les fonctionnalités de *snapshots* intégrées dans Petal permettent aussi d'avoir une vue cohérente du système.

4.2.4 GPFS (*General Parallel File System* - IBM)

GPFS[SH02] est le système de fichier connu pour être installé sur un des super-calculateurs les plus puissants du monde selon le classement du Top500 actuel : l'IBM ASCI White du *Lawrence Livermore National Laboratory* équipé de 512 processeurs. Les performances données sur l'ASCI White, sont à la hauteur du matériel mis en œuvre et permettent des E/S sur des fichiers à 7GB/s sur les 75TB de stockage.

GPFS est le nom du produit des laboratoires Almaden d'IBM qui est dérivé du *Tiger Shark File System*[HS96], système originellement destiné à servir des données multimédia à grande échelle comme, par exemple, pour la vidéo à la demande.

GPFS souhaite fournir une sémantique proche de la sémantique POSIX lorsque cela est possible. Les fichiers sont stockés sur plusieurs disques (striping) ce qui permet d'avoir de bonnes performances en accès. GPFS utilise des verrous distribués afin de synchroniser les lectures et écritures sur les disques, de manière à ne pas avoir de corruptions des données. Le système dispose de plus de moyens de reconnaître les modes d'accès classiques (lecture séquentielle, lecture à l'envers, ou lecture par pas) et d'agir en conséquence, en pré-chargeant dans ses caches les données devant être utilisées. Compte tenu des quantités de données en jeu, le système dispose d'un journal pour les méta-données mais les données ne sont pas journalisées.

[3]Ces fichiers (*sparse files*) sont obtenus par exemple avec une séquence de commandes du type : création puis écriture de quelques octets à l'offset 100000, fermeture. Si l'on a un stockage par blocs de 1000 octets il est intéressant de stocker uniquement le bloc (donc 4KB) qui contient les octets écrits et pas 100 blocs de zéros + 1 bloc avec les données.

Le système de verrouillage utilisé dispose d'un gestionnaire de verrou maître et de gestionnaires locaux (un par nœud). Le maître distribue une sorte de jeton de verrouillage qui autorise les sous-gestionnaires à effectuer les opérations de verrouillage sur un objet. Le sous-gestionnaire conserve le verrou tant qu'il n'est pas demandé par un autre client.

Deux sémantiques sont disponibles : la première utilisant un verrouillage par zones d'un fichier (*byte-range locking*) qui permet d'assurer une sémantique POSIX ; la seconde convient plus particulièrement pour les applications ne requérant pas une telle cohérence comme les logiciels utilisant MPI/IO. Dans ce second cas, le mode « rapide » (*data shipping*) est utilisé : les données sont réparties par blocs selon un algorithme de tourniquet sur plusieurs disques, ce qui permet ainsi d'éviter le surcoût consécutif à la gestion de la cohérence POSIX.

La gestion des méta-données de ce système reprend la gestion classique par inodes et blocs d'indirections. Un ajout utile est la gestion efficace des méta-informations : en effet, afin d'éviter une contention trop grande en cas de fichiers souvent ré-écrits sur le bloc de l'inode, plusieurs niveaux de verrous permettent à un système de cache de fonctionner. Par défaut, les inodes sont cachés sur les nœuds et un de ces derniers est désigné comme maître (*meta-node*) et ce sera lui seulement qui effectuera les écritures des mises à jour de l'inode. Par exemple, les opérations de mises à jour des dates d'accès en prenant ce verrou cumulatif en écriture et les mises à jour sont envoyées régulièrement des esclaves au maître qui va alors agréger les mises à jour. Si un client fait une opération nécessitant une synchronisation de ces informations (par exemple un appel `stat()`), cela va forcer la révocation du verrou et la synchronisation des informations afin que l'appel système fournisse les bonnes informations. Dans le cadre d'opérations plus complexes, un verrou exclusif sur l'inode peut être pris. Dans un mode de fonctionnement standard, les *meta-nodes* sont choisis selon un principe d'allocation sur le premier utilisateur, les utilisateurs suivants récupérant alors la référence du *meta-node* retenu.

GPFS offre de plus un certain niveau de tolérance aux pannes, mais cette fonctionnalité repose, pour une large partie, sur l'utilisation de RAID matériel sur les nœuds afin de limiter au maximum les problèmes pouvant survenir.

4.2.5 Conclusions sur les systèmes à disques partagés

Nous venons de présenter plusieurs systèmes utilisant un modèle de disques partagés. La plupart sont dédiés à des utilisation sur systèmes de stockage dédiés (SAN) ou bien émulent ce fonctionnement par des moyens logiciels (Frangipani/Petal). Nous venons donc de dresser un rapide aperçu de systèmes à disques partagés. Le développement de ces systèmes se poursuit encore actuellement et constitue une tendance forte des solutions de stockage. Le fait que les technologies SAN commencent à se répandre a aussi tendance à créer de nouveaux systèmes comme, par exemple, le sytème Lustre [CFS02][BZ01][Bra03] qui gère une séparation des données et des méta-données tout en

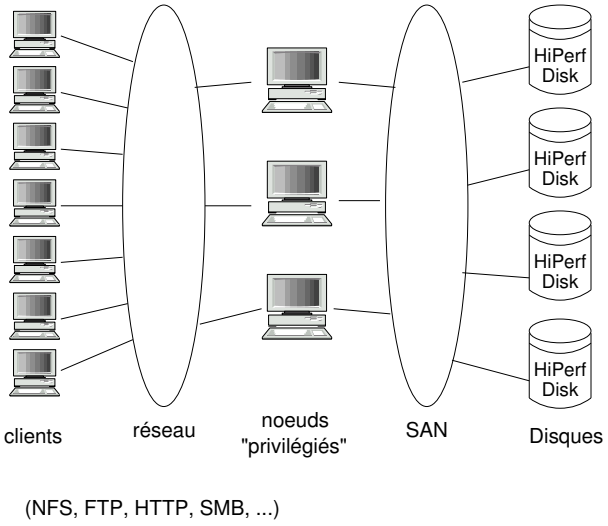

FIG. 4.1 – Ré-export de systèmes à disques partagés

l'enrichissant de diverses optimisations. Lustre utilise de plus un stockage orienté objet, c'est-à-dire que les disques disposent maintenant d'une certaine « intelligence » ; ces disques ne sont donc plus de simples périphériques de stockage car ils « comprennent » les données.

Pour résumer cette partie, nous dirons que les principaux points forts de ces systèmes sont donc de fournir :

– un découplage des serveurs et du stockage : si un nœud tombe en panne les données sont « encore là », éventuellement accessibles par un autre nœud
– un équilibrage de charge simplifié : n'importe laquelle des machines peut accéder à n'importe laquelle des données
– une vue logique unifiée de l'espace de stockage
– une extensibilité en capacité, connectivité, bande passante

Cependant, ces points forts ne vont pas sans laisser apparaître quelques défauts qui peuvent freiner une large adoption :

– coûts très élevés : matériel dédiés, compétences spécifiques, ce qui a tendance à rendre cette approche peu envisageable pour une simple grappe Beowulf…
– cohérence perdue si ré-export depuis plusieurs serveurs (c'est-à-dire les nœuds du système distribué) qui jouent alors le rôle de frontaux, si tous les clients ne peuvent pas être complètement considérés comme sûrs, comme cela est illustré sur la figure 4.1.

Sur cette figure, les serveurs intermédiaires accèdent de manière privilégiée aux disques du SAN et réexportent leur stockage vers les clients finaux par les protocoles « standards » tels que NFS, FTP, HTTP, SMB, … L'avantage de cette approche est de pouvoir multiplier les points d'accès au stockage et donc d'élargir le goulot d'étranglement que ces serveurs peuvent constituer.

4.3 Les systèmes de fichiers distribués

Ces systèmes sont distribués dans le sens où la gestion des données n'est pas assurée par un point unique. Nous allons donc présenter plusieurs des grandes familles existantes.

4.3.1 AFS et ses descendants (CODA, InterMezzo)

C'est une des plus anciennes familles de systèmes de fichiers distribués dont l'origine remonte à 1984 comme une partie du projet *Andrew* à l'Université de Carnegie Mellon (Carneggie Mellon University `http://www.cmu.edu/`)[4]. En 1989, la société Transarc est créée pour réaliser la commercialisation. En 1998, IBM acquiert Transarc puis redistribue 2 ans plus tard OpenAFS sous une licence *OpenSource*.

Plusieurs organisations (notamment le CMU et le CERN) utilisent ce système. Cette version est une version dérivée de la version commerciale (version 3).

Parallèlement, à partir de 1987, le développement du projet CODA a commencé comme un « branchement » de AFSv2 plus particulièrement orienté sur la gestion de la résilience.

Ce système a de nombreux points communs avec AFS mais il souhaite apporter des solutions plus efficaces liées principalement à un support avancé des modes « déconnectés », c'est-à-dire lorsque le client est déconnecté du réseau de manière volontaire (l'utilisateur s'en va) ou bien involontaire (le réseau devient indisponible).

Depuis quelques années, Inter-Mezzo est développé (originellement au CMU) et a des objectifs proches de ceux de CODA, ses auteurs le présentent comme une espèce de *re-design* et de *re-engineering* de CODA.

Il est à noter que des versions des clients sont disponibles pour de nombreux systèmes et architectures. Le code serveur reste cependant souvent destiné à des machines de types UNIX.

4.3.1.1 AFS

Une installation AFS se compose de deux types de machines : les serveurs qui hébergent les fichiers et les clients qui ne font qu'utiliser les dits fichiers.

AFS se compose de cellules (*cells*) et chaque cellule représente une zone d'autorité indépendante (à la manière des blocs d'adresses IPs ou des enregistrements DNS sur l'Internet).

Sur le client, ces cellules sont regroupées au sein de la hiérarchie AFS (`/afs/`) qui peut alors être intégrée dans le système de fichiers d'un client grâce à la VFS. Par exemple,

[4] Le 'A' de AFS signifie *Andrew* qui est aussi le prénom de Andrew Carnegie, co-fondateur de l'Université.

tout comme l'IMAG gère son domaine DNS `imag.fr`, l'Institut pourrait avoir une cellule `/afs/imag.fr` dont l'administration interne (découpage, nommage, etc...) pourrait être géré comme il le souhaiterait.

Les « sous-cellules » portent en fait le nom de « volumes » qui correspondent à des groupes de stockage de fichiers ayant des points communs (par exemple les fichiers d'un même utilisateur). En ayant des volumes pas trop grands, un équilibrage peut être effectué en répartissant ces volumes sur plusieurs serveurs (la question de leur localisation sera traitée un peu plus loin).

Un autre point qui mérite d'être précisé est que AFS a, dès le début, souhaité s'occuper des problèmes de sécurité et pour ce faire, utilise des techniques d'authentification mutuelle par le biais de Kerberos. Il est ainsi possible d'offrir un accès simplifié à un utilisateur distant, une fois que celui-ci a pu acquérir un *token* après authentification.

Un autre atout d'AFS pour les clients est une gestion de caches de fichiers performants. Le cache des fichiers distants sur disques locaux est apparu dès la version 2 du système et correspondait alors à des fichiers complets. La version suivante (version 3) a, entre autres, apporté un support pour un cache de morceaux de fichiers sur les clients, ce qui s'avère en général plus optimal pour les modèles usuels d'utilisation.

Le modèle de cohérence retenu est un modèle de cohérence à la fermeture (lors de l'appel `close()`) : une fois que le client relâche le fichier qu'il utilise, les mises à jour sont renvoyées vers la cellule responsable du fichier.

La séquence d'événements ci-dessous devrait permettre de mieux visualiser le théâtre des opérations :

1. L'utilisateur effectue un accès sur un fichier (`open()`).
2. Le gestionnaire de cache (par le biais du kernel) intercepte cet appel.
3. Il contacte un serveur qui gère la localisation des volumes pour trouver le bon serveur de fichiers.
4. Une fois la réponse reçue, les données sont récupérées et cachées par le gestionnaire de cache.
5. Le gestionnaire de cache renvoie une référence sur le fichier caché localement à l'application.
6. L'application effectue les entrées/sorties sur le fichier local.
7. L'application ferme le fichier (`close()`),
8. Le gestionnaire de cache (par le biais du kernel) intercepte cet appel et effectue le nécessaire pour resynchroniser la copie sur son volume originel.

Le gestionnaire de cache prend aussi soin d'assurer la cohérence de son cache en vérifiant que le serveur de fichiers ne lui casse pas une promesse de rappel (*callback promise*).

La sémantique de partage lors d'accès concurrents est gérée par une synchronisation vers la cellule hébergeant le fichier lors de l'appel système `close()`, que le client effectue lorsque l'application n'a plus l'utilité du fichier.

Architecture des serveurs Les machines AFS serveur doivent faire tourner un certain nombre de processus afin de simplement pouvoir assurer les services requis par les clients :

- serveur de fichier (*File Server*) : envoie les fichiers demandés et écrit sur disque les fichiers renvoyés par les clients
- serveur BOS (*Basic OverSeer Server*) : ce serveur vérifie le bon fonctionnement des autres serveurs et effectue les opérations nécessaires en cas de problème (arrêt / sauvegarde / relancement) et s'assure que des services incompatibles ne tournent pas en même temps (par exemple le serveur de fichiers ne devrait pas fonctionner en parallèle avec un outil à la `fsck` de récupération / réparation)
- serveur d'authentification (*Authentification Server*) : vérifie les *login/passwords* et authentifie les transactions. Ce serveur gère la base de données d'authentification.
- serveur de protection (*Protection Server*) : permet aux utilisateurs de gérer des ACL sur leurs fichiers. Pour ce faire, il gère une base de données de protection.
- serveur de volume (*Volume Server*) : gère les manipulations sur les volumes (migration vers une autre machine par exemple)
- serveur de localisation des volumes (*Volume Location Server*) : c'est la clé de la transparence pour les utilisateurs qui souhaitent accéder à leurs fichiers car il gère la base de données des localisation de volume (*Volume Location Database* - VLDB) qui assure comme son nom l'indique la correspondance entre le nom de volume et le nom de la machine qui l'héberge
- serveur de sauvegardes (*Backup Server*) : permet à l'administrateur de sauvegarder aisément une partie de l'arborescence
- *salvager* : permet de gérer la reprise après une panne du serveur de fichiers ou de volumes (l'équivalent de `fsck` lorsque l'on utilise un système de fichiers local)

Pour offrir un niveau accru de tolérance aux pannes, une base de données répartie est utilisée dans AFS : UBIK. Cette base offre en fait un fichier « plat » sur lequel peuvent être effectuées les opérations usuelles de fichiers (lecture / écriture / changement d'*offset*) mais le plus important est, sans doute, qu'elle offre un mode transactionnel qui permet de garder la base cohérente, c'est-à-dire que les mises à jour sont appliquées en entier exactement une fois ou bien, ne sont pas appliquées du tout. Un mode « lecture seule » peut être utilisé lorsque plusieurs entités UBIK partagent un même fichier. De plus, un algorithme d'élection permet à un serveur démarré de trouver sa place parmi les autres. Des votes ont aussi lieu pour accepter ou rejeter les mises à jour (en 3 phases : diffusion de la mise à jour, consensus et *commit* ou rejet). Il est à noter qu'UBIK requiert une synchronisation précise des différentes horloges afin de pouvoir fonctionner correctement (par le biais du protocole `NTP` par exemple).

La version estampillée OpenAFS correspond à une ouverture du code source de la version 3.6 de la version Transarc/IBM de AFS. Il existe une autre implantation des clients AFS, mais elle semble moins développée que OpenAFS.

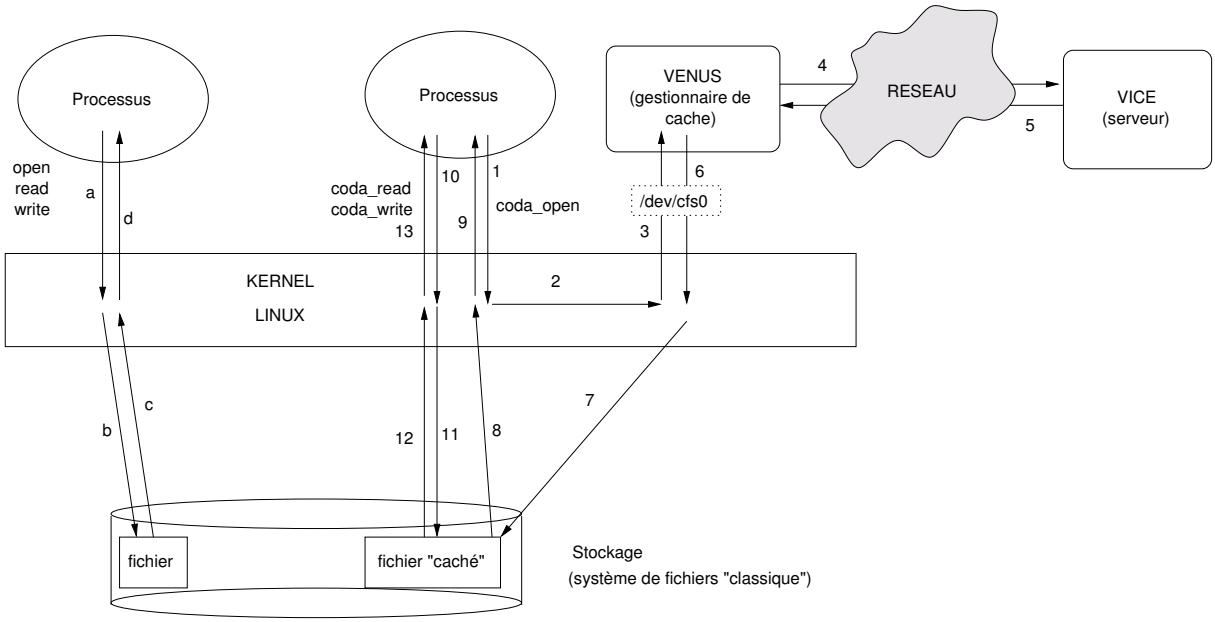

FIG. 4.2 – Architecture logicielle de CODA

4.3.1.2 CODA

CODA est actuellement encore développé au CMU par le groupe de Satyanarayanan et le support pour le client est intégré dans les noyaux Linux standards depuis 1997 (pendant la branche de développement 2.1.x).

Ce système hérite de nombreuses caractéristiques de son ancêtre (AFS2) notamment au niveau de la gestion de sécurité (le client potentiel n'est pas cru à moins d'avoir les crédits nécessaires) et de l'espace de nommage (par exemple le répertoire `/afs/` devient `/coda/`).

CODA est un système de fichiers à « gros grain » dans la mesure où il cache l'intégralité du fichier sur le disque local du client lorsqu'un accès est nécessaire. Une fois le fichier caché en local, il renvoie au client la référence du fichier local, ce qui permet d'avoir des accès performants sur les fichiers.

La sémantique de synchronisation est donc fixée par l'utilisation des appels systèmes `open()` et `close()`, c'est-à-dire que chaque client traite son fichier de manière indépendante et, quand il relâche la référence qu'il a sur le fichier, la réconciliation a lieu si possible. S'il ne peut y avoir de réconciliation automatique, un répertoire est créé et les fichiers conflictuels y sont déplacés pour que l'utilisateur (ou l'administrateur) règle le problème manuellement.

D'un point de vue architecture logicielle (cf. figure 4.2), le client CODA se compose de plusieurs entités, situées à différents niveaux :

– un support noyau : ceci est nécessaire afin d'avoir une intégration transparente dans le VFS. Le rôle de ce module est néanmoins simplifié car il sert simplement de « porte

d'entrée » dans le VFS, le gros du travail étant alors assuré par le démon VENUS, auquel une requête est envoyée par le biais d'un fichier *device* (`/dev/cfs*`)
– un démon de cache VENUS : ce démon gère le cache sur le poste client. Il est sollicité lorsque le client souhaite accéder à un fichier non disponible localement. Il va alors gérer le rapatriement d'une copie du fichier demandé. Dès lors le fichier sera caché dans le système de fichier local du client : les accès seront locaux. Lors d'un second accès, VENUS gérera la mise à jour du cache local auprès du serveur contenant le fichier maître. Ce démon assurera aussi le service des fichiers cachés localement lors d'opérations en mode déconnecté.

Afin de gérer les déconnexions fréquentes, les clients CODA peuvent disposer d'une liste de fichiers (*hoarded files*) ; ces derniers devront être récupérés et mis à jours automatiquement même lorsque qu'aucun accès n'y est fait. Typiquement, ceci est pensé pour les programmes de `/usr/` (pour simplifier l'administration de postes de travail par exemple) ou bien pour le compte d'un utilisateur « nomade », c'est-à-dire amené à être souvent déconnecté.

Le client se connecte donc de manière « indirecte » aux serveurs VICE puisque c'est VENUS qui va effectuer toutes ces opérations de manière transparente pour le client.

Une machine (éventuellement répliquée), SCM (*System Control Machine*) , doit assurer le rôle de « maître » et c'est elle qui va permettre d'offrir aux clients une abstraction du stockage dans la hiérarchie `/coda`. Les machines serveurs en quelques sortes « esclaves » disposent de trois services (pas forcément tous lancés sur une même machine) :

– serveur de fichiers (`codaserv`) : c'est la partie qui inter-agit avec les clients VENUS
– serveur d'authentification : le SCM maintient la copie principale et les autres serveurs disposent d'une copie en lecture seule
– serveur de mise à jour client : ce service s'assure que les copies des fichiers systèmes et des bases de données en lecture seule sont maintenues en cohérence avec leur contrepartie en lecture/écriture

Les serveurs de fichiers conservent leur propre vue du système (les fichiers sont identifiés par un numéro) et stockent le contenu des répertoires ainsi que les méta-données des fichiers (qui regroupent en plus des champs usuels des informations pour gérer le mode déconnecté, la réplication et le journal des opérations) dans un segment RVM (*Recoverable Virtual Memory*). Ce type de stockage, spécifique à CODA, permet de gérer des transactions sur les données stockées.

4.3.1.3 CODA pour les grilles : « *slashgrid - a framework for Grid-aware filesystems* »

Dans le cadre du projet Datagrid, le prototype *slashgrid* [McN02] a été développé sous forme d'une implantation différente du démon VENUS (pour CODA) afin de prendre en compte les spécificités Globus (authentification par certificats par exemple) et de pouvoir être compatibles avec la notion de comptes et d'identifiants d'utilisateurs (UID) alloués de manière temporaire par le système Globus.

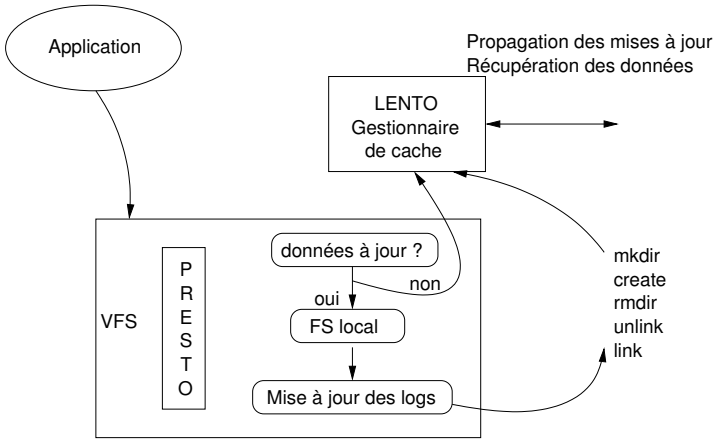

FIG. 4.3 – Architecture d'InterMezzo

Du point de vue du système local, l'utilisateur voit un système de fichiers de type CODA, localisé dans un `/grid` au lieu d'un `/coda` mais en coulisses, c'est le nouveau démon remplaçant le service VENUS qui gère tout cela.

4.3.1.4 InterMezzo

Le support pour ce système de fichiers [BN99] est entré de manière officielle dans le noyau Linux à partir de la version 2.4.15 du noyau (fin 2001).

D'après les auteurs, ce programme doit encore être fiabilisé, mais un sous ensemble de leurs objectif semble déjà être atteint. Tout comme CODA, avec lequel il a des similarités (un des instigateurs du projet, Peter J. Braam a travaillé dans l'équipe de Satyanarayanan, le père d'AFS), ce système vise a offrir de la haute disponibilité, principalement *via* la réplication de données (cache local persistent) et un support pour le travail en mode déconnecté.

Le but d'InterMezzo est de garder en cohérence des répliques de dossiers distribués sur plusieurs machines, les différentes copies étant maintenues en cohérence par le biais de *logs*, ces derniers étant envoyés aux autres détenteurs de copies. Un coordinateur par ensemble de fichiers joue le rôle d'organisateur/coordinateur pour les mises à jour. Après une déconnexion, il suffira d'envoyer les logs pour resynchroniser les copies.

InterMezzo utilise actuellement le système de fichiers disque `EXT3` (mais d'autres systèmes journalisés tels que ReiserFS ou SGI XFS sont envisagés) pour stocker ses données. À partir du moment où la partition est montée dans le VFS avec le type InterMezzo (c'est-à-dire `intermezzo`), ce dernier va intercepter tous les accès et veiller à gérer la concurrence des accès au stockage. De plus, en mode connecté la sérialisation des écritures peut être assurée, car le système peut fournir un verrou (*permit*) au client qui souhaiterait écrire.

La gestion du cache d'InterMezzo est assurée par 2 entités :

– PRESTO vit dans le noyau sous forme de module et joue le rôle de *wrapper* pour les accès au système de fichiers local qui joue aussi le rôle de cache
– LENTO vit en espace utilisateur et gère les opérations de contact avec les autres serveurs

Les premiers prototypes, développés sous forme de programmes en mode utilisateur sont disponibles sur le *web* à l'URL `http://www.inter-mezzo.org/`. Pour l'instant, le modèle de sécurité n'est pas encore complètement implanté et par ailleurs, le système ne dispose pas pas encore d'une réplication à la demande (pas encore implanté).

InterSync est un service utilisant InterMezzo pour pouvoir gérer la synchronisation avec systèmes de fichiers situés sur d'autres machines, sans avoir à parcourir complètement ces données (contrairement à ce que ferait `rsync`, par exemple). L'utilisation d'InterMezzo est intéressante de ce point de vue car l'utilisation des logs générés permet d'optimiser la synchronisation.

4.3.1.5 Conclusions sur la famille AFS

Ces systèmes bénéficient d'une longue histoire et de gestions performantes de cache, notamment par l'utilisation de caches sur disques. Le modèle de cohérence est de type cohérence de session ce qui peut convenir à la plupart des utilisations et permet d'avoir une gestion relativement efficace des caches.

Un aspect aussi, qui est souvent peu évoqué, est que bien que ne requérant de matériel spécifique, les données sont souvent stockées sur des gros serveurs de manière centralisée (notamment pour AFS et CODA) car le « grain » de répartition des données est souvent le volume qui constitue un gros morceau de hiérarchie de fichiers (par exemple un compte). Intermezzo est encore en développement actif et il lui manque actuellement le recul et la fiabilité de ses ancêtres.

4.3.2 Les systèmes à *logs* distribués

Ces différents systèmes (pas tous distribués) sont en fait des évolutions et nouvelles branches développées à l'université de Berkeley dès fin 1984 par l'équipe d'Ousterhout. À l'origine, le projet Sprite souhaitait offrir des solutions pour les réseaux locaux qui, à l'époque, apparaissaient comme très prometteurs, notamment en ce qui concerne la gestion des ressources de calcul, de stockage[5], ...

Un des buts consistait en la réalisation d'un système distribué et transparent (*Single Image System*) pour un ensemble de machines au sein d'un réseau local et mettait en place une distribution avancée des données et des processus (migration par exemple).

À partir de 1991, l'équipe a commencé a souhaiter mettre un terme au projet qui devenait alors trop important et complexe pour les ressources disponibles. L'arrêt définitif des développements a eu lieu quelques années plus tard.

[5]Pour rappel, le protocole NFS, maintenant omniprésent, n'existait même pas encore !

4.3.2.1 Zebra (*The Zebra Striped Network Filesystem*)

Ce système développé à partir de 1992 a rejoint le système Sprite. L'idée dominant ce projet a été de distribuer des morceaux de fichiers sur plusieurs serveurs (comme ce qui se produit dans les systèmes RAID).

Afin de bien assimiler le fonctionnement de Zebra, il faut évoquer le système de fichiers (local) qui l'a précédé au sein du projet Sprite : LFS [OD89][RO92]. Ce système correspond à la seconde version du système de fichiers de Sprite (début des années 1990). Il a été pensé pour diminuer les opérations mécaniques sur les disques, tout en optimisant le placement des données. En effet, bien que la bande passante des périphériques de stockage ait fait des progrès (disques plus rapides, technologies de type RAID), les temps d'accès au médium (latence) ont certes été améliorés mais à une échelle moindre.

Le système utilise un journal sur le périphérique auquel il ajoute les données à écrire à la fin. En parallèle, un système compacte ce journal, c'est-à-dire qu'il choisit des morceaux de ce journal et agrège les espaces libérés ou les fichiers fragmentés. Une fois le périphérique rempli, les données sont écrites dans les « trous » du journal.

Afin d'avoir des performances, un index des fichiers écrits dans le journal est conservé pour permettre un accès direct aux inodes écrites au fur et à mesure du fonctionnement du système donc directement dans le *log*. Ces inodes fonctionnent de la même façon que les inodes classiques UNIX et gèrent plusieurs niveaux de redirections ; la seule différence est que les inodes ne sont pas regroupés dans une table d'inodes mais écrits au fur et à mesure dans le journal, ce qui forve LFS à maintenir une table des inodes à jour pour ne pas avoir à parcourir tout le journal, à chaque demande d'accès.

LFS a donc permis de développer les concepts de *log* et ceux-ci ont eu une application peu de temps après dans le système de fichiers Zebra[HO01] pour Sprite.

En effet dans ce système de fichiers distribué qui mêle redondance et striping, les logs sont utilisés à plusieurs endroits. Afin de mieux apprécier l'architecture du système, décrivons les différents composants ainsi que leur(s) rôle(s) :

- un gestionnaire de fichiers qui gère les méta-données (hiérarchie des fichiers) : les méta-données sont elles mêmes des fichiers Sprite qui contiennent toutes les informations sur les données (localisation des blocs de données sur les serveurs de stockage) et permettent de répondre aux diverses demandes des clients (*lookups*). Lorsqu'il détecte qu'un fichier est ouvert par plus d'un nœud, il invalide et désactive le fonctionnement des caches sur les clients afin de garantir la cohérence des données.
- plusieurs serveurs de stockage : ceux-ci stockent des morceaux de 512KB avec une structure de *log* ; pour récupérer l'espace rendu disponible au fur et à mesure de l'utilisation du système, ils sont contactés par le nettoyeur.
- un nettoyeur de *stripe* qui permet récupère l'espace libéré : la structure en *log* des fichiers fait que les nouvelles données ne sont pas des modifications des anciennes mais des nouvelles ajoutées au *log*. Par conséquent, au bout d'un moment, la place doit être récupérée et c'est le rôle de ce processus qui doit identifier les blocs encore « vivants »

– plusieurs clients : chacun des clients produit des données qui sont stockées sous forme de *log*, et c'est ce dernier qui va être *stripé* sur plusieurs serveurs de stockage (après calcul du bloc de parité). Lors d'une lecture, le gestionnaire de fichiers est d'abord interrogé pour récupérer la localisation des données à lire puis ensuite le client transmet ses requêtes aux serveurs de stockages. Lors d'une écriture, les données sont placées dans un cache local et envoyées au serveur lorsqu'un des événements suivant se produit : expiration d'un *timeout*, remplissement du cache, synchronisation explicite, demande d'invalidation du cache du gestionnaire de fichiers pour gérer la cohérence des caches des différents clients.

Les blocs contiennent soit des données brutes, soit des *deltas* (créés lors d'ajouts ou de modifications de blocs dans un fichier par l'application ou par le nettoyeur). Ces deltas contiennent alors un identificateur et une version du fichier, ainsi que le numéro du bloc, et des références sur l'ancienne et la nouvelle version du bloc (afin de gérer le cas où celui-ci aurait déjà été remplacé).

4.3.2.2 xFS[6] (*A Serverless File System* - Projet *Berkeley NoW*)

En 1993, le projet NoW (*Network of Workstations*) naît des cendres du projet Sprite et souhaite maintenant mettre en place une approche « sans serveur » dans laquelle chaque machine participant au système est en mesure de stocker, contrôler, cacher et accéder aux caches distants.

Les premiers choix architecturaux concernant xFS[WA93] ont principalement consisté en une mise en œuvre hiérarchique des machines stockant des méta-données, et ce, afin d'améliorer les performances du système dans le cadre de stockage à grande échelle (les communications sur WAN s'avérant coûteuse - au moins - en terme de latence).

Le projet s'est cependant recentré, quelques années plus tard, sur une utilisation sur réseaux locaux [ADN+95]. Son architecture globale reprend plusieurs concepts développés précédemment, tout en les optimisant :

– système de *logs* et de *striping* à la Zebra : les modifications ont consisté en un retrait des goulots d'étranglement que constituait le processus nettoyeur (unique dans Zebra) et le serveur de fichiers (aussi unique dans Zebra)
– gestion des caches coopérative : xFS met en œuvre un système de caches en mémoire distribué et coopératif. Cette approche apporte bien évidemment de nombreux avantages, car la lecture d'une donnée dans un cache mémoire distant au travers d'un réseau rapide est très souvent plus performante que sur un disque local (la latence réseau est plus faible que la latence disque). Le niveau de cache a aussi vu son grain diminuer dans Zebra, en passant du fichier au bloc et ce, afin d'améliorer les performances.

xFS ayant pour *leitmotiv* « *anything, anywhere* », toutes les données et méta-données peuvent donc se trouver sur n'importe lequel des nœuds et même être migrées en cours de fonctionnement.

[6] A ne pas confondre avec le système de fichiers journalisés développé par SGI : XFS.

Un des inconvénients de cette approche est qu'un tel système requiert une interaction poussée avec le système de gestion mémoire du système d'exploitation qui l'héberge. De plus les machines doivent comme dans le cas (plus récent) des accès mémoires distants (par RDMA par exemple) se faire une totale confiance, car elles partagent toutes un accès total au système. Pour pouvoir offrir des données à des clients plus « douteux », il faut donc ré-exporter le système xFS par un autre moyen comme on le ferait dans le cadre de systèmes à disques partagés (comme cela a déjà été présenté en section 4.2 page 48).

4.3.2.3 Conclusion

Les approches de ces systèmes sont très intéressantes mais le problème est qu'ils sont très souvent très intégrés au système d'exploitation et donc, difficilement et raisonnablement portables. Des adaptations de ces concepts (*logs* et distribution) à des environnements plus récents pour Linux ont par exemple été menées récemment au sein des projets Swarm/Sting [HMS99][MH00] de l'université d'Arizona mais les développements ont été arrêtés depuis.

4.3.3 PVFS (*Parallel Virtual File System*)

Cette partie va présenter le système PVFS (dans sa première version). Actuellement une version 2 est en cours de développement et est une complète refonte du système existant. Nous présenterons rapidement les buts recherchés et les choix déjà effectués pour cette future version.

4.3.3.1 PVFS1

Ce système [CIRT00][RCLL02] a commencé à être développé à partir de 1996 au laboratoire PARL[7] de l'Université de Clemson et continue toujours d'être maintenu. Plusieurs aspects de ce système sont intéressants :

- il permet d'utiliser les ressources disques de plusieurs machines afin de présenter à l'utilisateur une partition virtuelle dont la taille correspond à la somme des tailles des disques ;
- l'utilisateur peut préciser - s'il la connaît - la répartition de ses données (*striping*) qui sera utilisée par PVFS ;
- ROM-IO peut être compilé avec un support PVFS et offre alors une interface compatible avec MPI-IO.

Un système PVFS est constitué de trois entités logicielles qui peuvent être installées de manière indépendante : un *manager* (`mgr`), des démons d'Entrées/Sorties (`iod`) et des nœuds clients. Ces entités communiquent bien sûr entre elles et ce, grâce à des canaux

[7] Voir Site Web de PVFS `http://parlweb.parl.clemson.edu/pvfs/`.

TCP. Dans un système PVFS, il doit y avoir exactement un `mgr` et au moins un `iod`. En général, plusieurs `iods` seront utilisés afin de répartir la charge des E/S disque et réseau sur plusieurs machines physiques (bien que plusieurs `iods` puissent cohabiter sur une même machine).

Nous allons maintenant décrire de manière un peu plus détaillé le rôle qui est assuré par chaque programme :

– Le `mgr` est aussi appelé serveur de méta-données : c'est lui qui reçoit les requêtes des clients et qui les met en relation avec les iod qui contiennent les données demandées.
  Lorsqu'un client souhaite accéder à une donnée, il transmet une requête au `mgr` contenant des informations sur le fichier et l'*offset* des données qui doivent être récupérées. Le `mgr` consulte alors ses méta-fichiers et renvoie alors au client les informations qui vont lui permettre de contacter les `iods` qui stockent ses données. Le transfert s'établira alors entre le client et le(s) `iod(s)` qui le concernent. Il n'y maintenant aucun cache des méta-données sur les clients alors que les premières versions utilisaient des exports NFS pour partager les données[LR99].
– Les `iods` sont utilisés uniquement pour effectuer les stockage physique des données. Ces serveurs attendent les requêtes des clients, et lisent (ou écrivent) sur disque les données devant être manipulées. Les blocs de données sont stockés sous la forme de fichiers dans un système de fichier Linux standard (`EXT2` par exemple). Le nom du fichier est déterminé par un *hash-code* calculé en fonction de divers paramètres du fichier et ce, afin de pouvoir accéder rapidement aux fichiers contenant les données.
– Les clients sont les utilisateurs du système PVFS. À la manière de clients NFS, ils ne connaissent explicitement que la localisation du `mgr` (hôte, port et nom du répertoire exporté). Les clients « montent » la partition PVFS exportée dans leur VFS Linux grâce à un module noyau dédié qui déclare le type `pvfs`. Celui-ci assure uniquement l'intégration au sein du VFS et communique avec un démon `pvfsd` fonctionnant en mode utilisateur. C'est ce démon qui servira d'intermédiaire avec les iods. Depuis peu (version 1.5.6), une version du démon `pvfsd` peut être intégrée dans le noyau afin d'éviter les pertes de performances dues aux va-et-vient entre espace utilisateur et espace noyau.

Étant considéré comme un nouveau système de fichiers, la mise en place d'un système PVFS sous Linux requiert le chargement d'un module noyau d'interfaçage des fonctions PVFS avec le noyau sur les clients.

La tolérance aux pannes n'est pas gérée mais divers travaux d'ajouts de redondance sont menés : par exemple, CEFT-PVFS[ZJQ+03] consiste à mettre un place une réplication de type RAID-10 en répliquant toute l'architecture PVFS (méta-données et données). Dans la version originale de PVFS, un mode de gestion de redondance paresseuse (*lazy redundancy*) devrait permettre de calculer des sommes de parités à des moments définis par l'utilisateur.

Les développements actuels se concentrent sur l'ajout de fonctionnalités de *monitoring* sur le serveur mais la plupart de l'équipe développant PVFS se concentre sur la deuxième version du système.

4.3.3.2 PVFS2

PVFS2 est une refonte totale du système et souhaite offrir plus de souplesse, de modularité et d'extensibilité. Les nouveautés envisagées doivent toucher à plusieurs points critiques et on peut notamment déjà évoquer :

- distribution des serveurs de méta-données (PVFS1 : 1 seul méta-serveur)
- support de plusieurs protocoles (PVFS1 : seulement TCP)
- support de plusieurs périphériques de stockage et de méthodes d'accès
- support de *plugins* pour la distribution des fichiers
- sémantique de cohérence à la demande
- optimisations (*multi-threading* des applications et meilleur *scheduling* de la gestion des requêtes)

Au moment où nous écrivons, il n'y a pas encore de prototype disponible de cette version et l'équipe de développement continue à améliorer la version précédente.

4.3.4 SFS (*Self-certifying File System*) et projets dérivés

SFS[Maz00] a été conçu avec trois buts principaux :

- sécurité : une hypothèse de fonctionnement est que le réseau serait aux mains de tiers malicieux[8] susceptibles de retarder ou de cacher l'existence de serveurs jusqu'au rétablissement du mode de fonctionnement normal ;
- espace global de noms : tout est stocké et accessible depuis la hiérarchie montée dans `/sfs`;
- contrôle décentralisé : il n'y a pas besoin d'autorité centrale à contacter pour démarrer un serveur SFS visible depuis l'Internet.

4.3.4.1 SFS

Ces buts sont atteints en séparant la gestion des clés de la gestion de la sécurité du système de fichiers. Ce résultat est obtenu avec des chemins de fichiers auto-certifiés, c'est à dire que le nom contenu dans le chemin permet d'authentifier le serveur sans avoir recours à un mécanisme supplémentaire. Ainsi sous SFS, les chemins ont une forme `/sfs/hote:host_id` avec `host_id` = *hash*(`hôte`, clé publique du serveur)

La fonction de *hash* choisie doit avoir de bonnes propriétés de résistance aux collisions et actuellement c'est l'algorithme SHA[9] qui est utilisé.

Le partie client SFS joue en fait le rôle d'un serveur NFS pour le client NFS. C'est elle qui va, en quelque sorte, gérer les transferts avec le serveur SFS qui, lui, va jouer le rôle d'un client NFS pour le serveur NFS, comme cela est illustré sur la figure 4.4 page suivante.

[8] Je suspecte les auteurs de ce projet d'avoir eu à faire avec les personnages décrits dans [Tra01] :-)

[9] Cet algorithme développé par le NIST génère un *hash* de 160bits pour les messages qu'il reçoit en entrée et est aussi utilisé dans de très nombreux autres logiciels (FreeNet, BitTorrent, GPG, ...)

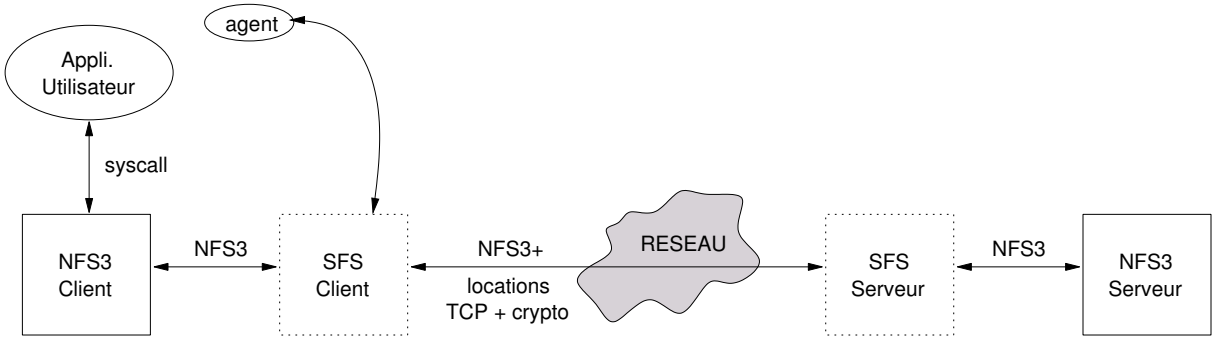

FIG. 4.4 – Architecture SFS

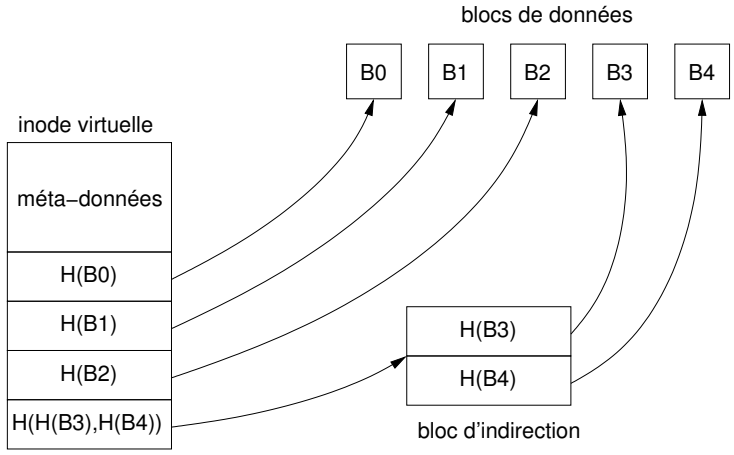

FIG. 4.5 – Utilisation d'arbres de *hashes* pour retrouver les blocs de données

4.3.4.2 SUNDR (*Secure Untrusted Data Repository)*

Ce projet débuté par David Mazières mélange un aspect système de fichiers ainsi qu'un aspect pair-à-pair et aussi un aspect cryptographie.

SUNDR[MS02] détaille un système permettant d'utiliser des serveurs de stockages é-ven-tuel-lement peu sûrs. L'utilisateur signe ses mises à jour, et les données stockées sont repérées et validées par leur somme de *hash*. L'utilisation d'un *hash* fort (actuellement SHA-1) offre, en particulier, de bonnes garanties que les blocs de données lues n'ont pas été modifiés.

En *hashant* l'inode virtuelle, un handle sur le fichier peut-être obtenu. Pour mettre à jour un fichier, le client a simplement à stocker les blocs de données mis à jour (ces derniers auront des *hashes* différents) et mettre à jour les *hashes* jusqu'à remonter à l'inode virtuelle. Le client doit alors signer cette nouvelle version du fichier et la fournir au serveur gérant ces inodes afin de vérifier l'authenticité de la mise à jour. Si le client arrive à lire les données, il est assuré qu'elles n'ont pas été corrompues.

4.3.4.3 CFS (*Cooperative File System*)

CFS [DKK+01] décrit un système de fichiers en lecture seule qui s'appuie sur une architecture pair à pair. De même que dans SUNDR, ce système met en place des arbres de *hash* et développe des techniques similaires aux arborescences des systèmes de fichiers locaux pour gérer les répertoires et les données (en remplaçant les références sur des endroits du disque par des sommes de *hash*). Lorsque qu'un bloc doit être recherché à partir de son *hash*, une architecture de type CHORD permet de retrouver rapidement le bloc correspondant au *hash* demandé.

4.3.5 Récapitulatif sur les systèmes de fichiers distribués

La liste des systèmes que nous venons de présenter est loin d'être exhaustive mais elle permet à notre avis d'avoir une vision assez variée de plusieurs systèmes et architectures existantes.

De plus l'intégration plus ou moins aisée de bibliothèques d'accès dans le VFS par le biais de modules de systèmes de fichiers en mode utilisateur (sous Linux ou bien sous forme de « traducteurs » dans certains OS à micro noyau) augmente de manière considérable le nombre potentiel des systèmes disponibles distribués.

C'est ainsi que tous ces systèmes ont des buts plus ou moins différents et donc, logiquement, des caractéristiques associées qui s'adaptent plus ou moins bien aux grappes *Beowulf*.

4.4 Conclusion

Nous venons donc de dresser un panorama (loin d'être exhaustif !) qui présente plusieurs des systèmes principaux (exports, modèles à disques partagés, modèles à messages) que nous avons observés dans le cadre de nos recherches, soit au travers de la littérature (pour les modèles avec des besoins particuliers tels que le matériels dédié), soit au travers d'expérimentations plus pratiques (pour entr'autres CODA, AFS, PVFS,...) Nous avons donc par la suite trié ces systèmes selon plusieurs critères afin d'avoir une vision assez large des différentes fonctionnalités que ceux-ci offrent, tout en pouvant effectuer un choix parmi la grande quantité de solutions disponibles.

Tout d'abord nous avons observé quelques systèmes d'export : ils ont une approche relativement simple et intuitive, car ils sont intrinsèquement centralisés et, comme leur nom l'indique, ils exportent un système de fichiers local. Le problème de cette approche est que la centralisation pose des problèmes de passage à l'échelle mais, en revanche, ce sont les systèmes les plus répandus actuellement.

Les systèmes à disques partagés offrent souvent de très bonnes performances mais requièrent souvent l'utilisation de matériel dédié (baies SAN pour la plupart, voire

disques spéciaux pour GFS). Disposant ainsi de bonnes performances matérielles, ils peuvent souvent se permettre d'assurer une cohérence forte des systèmes voire même une cohérence complète UNIX. Cependant le coût de déploiement est souvent prohibitif et ils requièrent une totale confiance en les nœuds clients, car, si un des nœuds veut faire des bêtises, il le peut pusiqu'il a un accès total aux disques.

Les systèmes distribués à messages sont d'une part beaucoup plus nombreux et d'autre part ont des approches très différentes les uns des autres. Nous avons essayé de présenter une variété de ces systèmes disponibles qui pourraient être utilisés pour offrir du stockage. Les hypothèses d'utilisation de tous ces systèmes divergent donc, et différents modèles se côtoient notamment au niveau de :

– la sécurité : va du mode « paranoïaque » (systèmes à la AFS ou SFS) à un mode plus simple (permissions Unix)
– la cohérence offerte : sémantique de session (AFS), sémantique temporelle « au petit bonheur », pas de cache (PVFS), ...
– distribution des données et méta-données allant du « *anything anywhere* » de xFS à une centralisation de certains services comme dans Zebra ou PVFS ou bien totale dans les systèmes d'exports (NFS / CIFS)

Le tableau 7.6 page 120 vise à effectuer une synthèse des différents critères que nous avons présentés en section 2.4 page 28.

Souhaitant disposer d'un système standard, fonctionnant sur du matériel générique (*commodity hardware*) et se présentant sous la forme de briques de bases, nous nous sommes tournés vers les systèmes utilisant NFS. Ce choix peut paraître surprenant dans la mesure où le modèle de cohérence est flou (cohérence temporelle) mais en revanche sa disponibilité de base sur la majorité des systèmes Unix existants nous a fortement poussé dans cette direction. Diverses approches ont été réalisées par le passé et certains développements continuent à être réalisés autour de ce protocole afin d'en améliorer les performances. Nous en présenterons plusieurs dans la partie suivante (plus particulièrement dans la section 5.2 page 72).

Nom	Environnement				Fonctionnalités			Pérennité	
	Intrusivité	Pré-requis	Linux	API	Données	Méta-données	Cohérence	État	Licence
NFS	0	0	*	++	0	0	+	++	S
CIFS	0	0	*	++	0	0	++	++	O/P[10]
CIFS/DFS	0	0	-	+	+	+	++	?	P
MFS	++	0→++ (Mosix)	*	++	+	+	++	++	O
CXFS	++	+++ (SAN)	C+S ?	+	++	++	+++	++	P
GFS	++	+++ (SAN)	*	+	++	++	+++	++	P/O[11]
Petal	++	++ (réseau)	0	+	++	++	+++	?	P
GPFS	++	+++ (SAN)	C+S ?	+	++	++	+++	++	P
(Open)AFS	++	serveurs dédiés	*	++	+	+	++	++	O
CODA	++	serveurs dédiés	*	++	+	+	++	+	O
InterMezzo	++	?	*	+	+	0	++	+	O
Zebra	++	serveurs dédiés	0	+	+	0	+++	-	O
xFS	++	serveurs dédiés	0	+	++	++	+++	-	O
PVFS	+->++	non	*	++	++	0	0	++	O
SFS	0	non	*	+	0	0	+	?	O
SFS/SUNDR	0	non	*	+	?	?	+	?	?
SFS/CFS	0	non	*	+	++	?	+	?	?

TAB. 4.1 – Récapitulatif des systèmes de fichiers distribués
Les critères ont été définis dans le tableau 2.1 page 30.

Chapitre 5

Présentation de la proposition NFSP

Ce chapitre a pour but de préciser les contraintes qui ont guidé la conception de notre prototype, NFSP, dont entre autres une intrusivité minimale : les clients devant rester tels quels. Nous présenterons aussi d'autres systèmes et projets utilisant NFS d'un manière distribuée afin d'en améliorer les performances. Nous présenterons ensuite notre proposition, en précisant dans quelle mesure les contraintes que nous nous étions fixées nous ont éloignés des solutions existantes. Par la suite, nous présenterons donc les divers choix d'implantation que nous avons été amenés à mettre en place, en les justifiant, par rapport aux applications que nous envisagions.

5.1 Introduction/Historique

En partant à la recherche d'un système de fichiers pour la grappe *i-cluster*, nous avons commencé par étudier quels systèmes seraient envisageables pour ce système de 225 nœuds (voir l'architecture détaillée en Annexe B page 153). Les machines n'étant pas à priori conçues pour faire du calcul intensif, car, étant destinées de par les choix ayant guidé leur conception à un usage bureautique, cela restreignait beaucoup les possibilités envisageables, et la plupart des systèmes présentés dans les parties précédentes ne pouvaient donc pas être utilisables. De plus, comme nous l'avons présenté dans nos critères, un pré-requis fort présent, dès le début, a été de fonctionner avec un système Linux, la grappe devant utiliser ce système d'exploitation. De plus, pour pouvoir bénéficier de transferts optimisés entre deux grappes (qui était une de nos applications principales), il fallait que les données soient distribuées sur plusieurs machines pour pouvoir multiplier les interfaces « émettrices ».

En cette année 2000, l'installation des machines étaient alors effectuées de manière plus ou moins « artisanale » avec les outils alors disponibles (*boot* par PXE, clônages d'images de disques) ou bien en cours de développement [ABDM01][ADM01]. Une fois l'installation terminée, une machine un peu plus puissante que les autres jouait alors le rôle de serveur NIS et de serveur de fichiers en utilisant le protocole NFS.

Après l'étude de plusieurs systèmes que nous avons présentés dans les chapitres précédents, nous nous sommes plus particulièrement intéressés à la famille AFS mais l'infrastructure requise pour une telle installation nous avait semblé un peu trop coûteuse humainement (installation et maintenance) et d'autre part, l'environnement sécure utilisé par ces familles était à notre avis superflu pour une grappe déjà protégée du monde extérieur. De plus, les données étaient par défaut réparties par gros grain (fichier). Nous nous sommes donc intéressés à PVFS, mais à l'époque (courant 2001) les tests préliminaires que nous avions menés ont montré de nombreux problèmes de stabilité sur les clients, ce qui n'a pas contribué à rassurer les administrateurs quant à la disponibilité des machines ni à leur laisser une tranquillité d'esprit :-) En effet, si un client plantait, il fallait bien souvent relancer une bonne partie des machines clients et les serveurs de données du système PVFS. Un module spécifique devait être chargé sur les clients ainsi qu'un démon associé, ce qui le rendait un peu trop intrusif à notre goût. Les données étaient cependant réparties sur plusieurs serveurs, ce qui constituait un aspect intéressant pour les transferts grappe à grappe que nous envisagions.

C'est cet ensemble d'aspects qui nous a orientés à observer les solutions disponibles en NFS conjuguant *striping* des données, « zéro modification » sur le client et utilisant du matériel standard relativement peu performant. De plus, pour être adopté, il fallait que le tout tourne sous Linux et soit disponible sous une licence permettant de modifier aisément les prototypes afin de pouvoir expérimenter diverses extensions (redondance, etc).

Nous présentons donc dans la section suivante plusieurs développements réalisés autour de NFS - il faut cependant noter que certains de ceux-ci n'étaient pas encore disponibles au moment où nous avons commencé nos travaux sur NFSP.

5.2 Autres projets utilisant NFS

L'omniprésence de NFS dans le monde des différents UNIX a entraîné de nombreux travaux l'utilisant dans divers rôles et sous des architectures logicielles plutôt variées. Cette section vise à présenter plusieurs de ces projets et leurs liens avec l'utilisation de NFS.

5.2.1 NFS est parfois utilisé comme un protocole de « glue »

NFS peut être utilisé pour « rentrer » dans la VFS et ainsi implanter un système de fichier en mode utilisateur. Le serveur utilise alors l'interface `loopback` comme *front-end* et peut utiliser le *back-end* de stockage qu'il souhaite. C'est ainsi que divers systèmes ont permis de faire rentrer beaucoup de choses dans la VFS comme par exemple le projet Gecko [BH99] qui offre un *proxy* web ou bien un autre projet [GS02] qui lui permet de jouer le rôle de *proxy* FTP et tout cela par simple montage NFS sur les clients.

Il existe bien sûr d'autres méthodes pour « entrer » dans le VFS en utilisant d'autres systèmes, par exemple des résultats similaires peuvent être obtenus en émulant le fonctionnement d'un serveur VICE comme le font `podfuk[Mac]` et `slashgrid`[McN02].

Il existe aussi pour Linux des systèmes dédiés à la résolution de ce problème et qui permettent d'implanter un système de fichiers en mode utilisateur. Parmi les projets récents, peuvent être cités AVFS (*A Virtual File System*) et FUSE (*Filesystem in USErspace*).

D'une manière plus générale, un module implanté en mode noyau permet de re-router les appels effectués par le VFS vers un processus utilisateur. Ce sont souvent des mécanismes d'*upcall/downcall* au travers d'un fichier spécial *device* (`open()` / `read()` / `write()` / `ioctl()`) qui sont utilisés, mais les autres moyens de communications offerts par le noyau peuvent être utilisés, voire abusés, comme par exemple la hiérarchie `/proc`...

Dans le domaines des systèmes de fichiers plus « classiques », SFS (voir la section 4.3.4, page 66) utilise le protocole NFS pour d'une part entrer dans le système de fichiers virtuels et d'autre part pour transférer les requêtes entre le client SFS et le serveur SFS.

5.2.2 Avec des répartiteurs de charges entre les clients

Plusieurs systèmes consistent à installer une machine qui va jouer le rôle de frontal pour les clients, cette machine routant alors les requêtes vers les serveurs qui peuvent les gérer.

5.2.2.1 NFS^2

Dans NFS^2 [Mun01], le code des serveurs et des clients n'est pas modifié et plusieurs serveurs sont agrégés par un répartiteur de charge comme illustré sur la figure 5.1 page suivante.

Les serveurs sont utilisés comme unités de stockage de fichiers, et NFS^2 sert donc à offrir un unique espace de nommage. Les fichiers peuvent être préférentiellement stockés sur des sous serveurs spécialisés qui, par exemple, disposent de solutions de haute disponibilité.

Le principe de fonctionnement de ce système est que les *handles* NFS contiennent des informations que le répartiteur comprend, ce qui lui permet de gérer la répartition entre les différents serveurs NFS de stockage. Un répertoire et ses fichiers est stocké sur une partition P mais les sous répertoires ont de grandes chances d'être stockés sur d'autres partitions, ceci étant fait afin de pouvoir mettre en place une répartition de la charge.

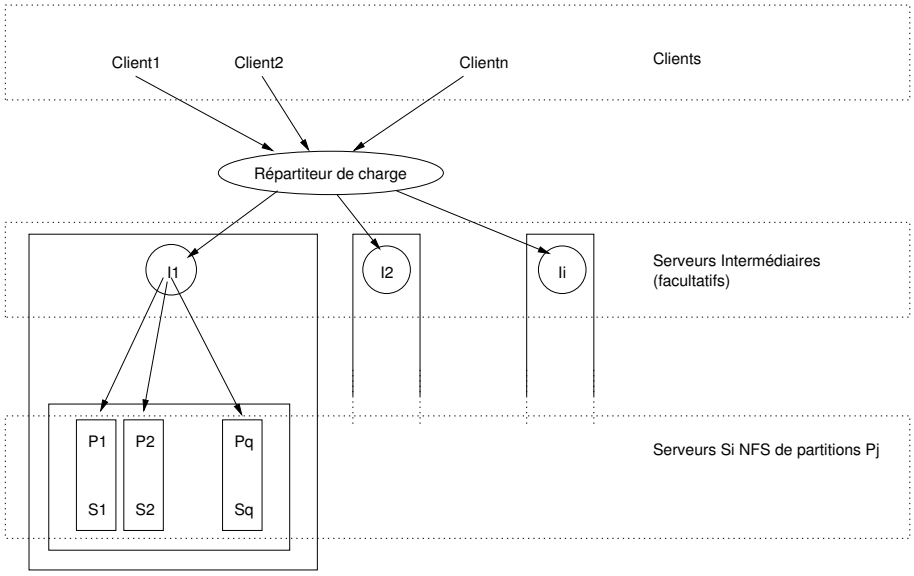

FIG. 5.1 – Le système NFS^2

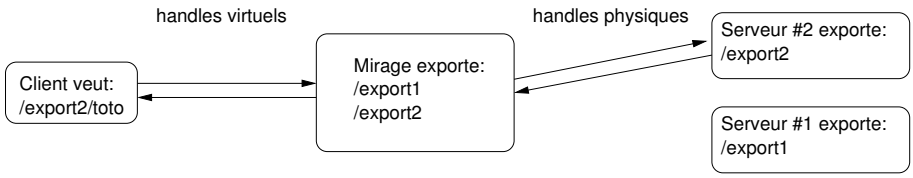

FIG. 5.2 – Routeur NFS Mirage

5.2.2.2 Routeur NFS Mirage (*Mirage NFS Router*)

Ce système[BH02] permet d'agréger plusieurs serveurs NFS « normaux » sous la forme d'un serveur virtuel. Les clients ne dialogueront qu'avec le serveur virtuel qui effectuera la conversion des handles utilisés entre les clients et le serveur virtuel et ceux utilisés entre le serveur virtuel et les serveurs « de stockage ».

L'unité de séparation est un serveur NFS et il n'y a donc qu'une répartition à gros grains possible ; le serveur virtuel fonctionne donc un peu comme un ré-export de montages NFS et donc une concaténation des exports. Il est à noter qu'un autre mode peut être envisagé et consisterait à faire l'union des contenus de `/export1` et `/export2` (voir figure 5.2) mais dans ce cas-là, le comportement à adopter en cas de créations de fichiers ou répertoire n'a pas une solution évidente.

L'approche du serveur est aussi de pouvoir servir à protéger un serveur des attaques de type « déni de service » en établissant un premier filtrage intelligent, avant de transmettre les requêtes aux vrais serveurs.

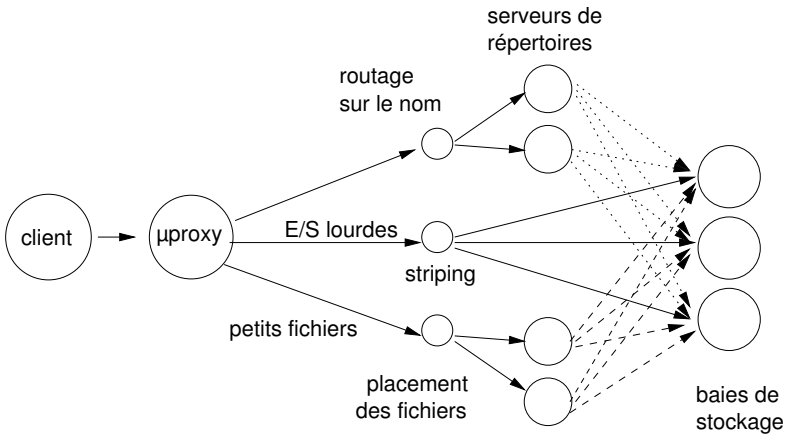

FIG. 5.3 – Fonctionnement de Slice

5.2.2.3 Slice et son μproxy

Slice [ACV00] (2000) est un système de stockage dédié à une utilisation sur réseaux rapides, qui interpose un filtre de paquets appelé μproxy sur les clients. Le code client est conservé mais un filtre de paquets est installé dans sa pile réseau.

Un serveur virtuel est défini comme destination par exemple une adresse IP « inexistante » et lorsque le μproxy reconnaît des requêtes NFS à destination de ce serveur, il les transforme puis les route vers le serveur adéquat selon leur type comme cela est illustré sur la figure 5.3.

Trois classes de serveurs sont utilisées comme destinations des requêtes ré-écrites : la première gère les opérations liées aux répertoires, la seconde correspond aux E/S lourdes (lectures/écritures) et la troisième gère les E/S sur les petits fichiers (en fait les premiers 64KB au début des fichiers). En agissant ainsi à bas niveau comme filtre, le μproxy peut éventuellement « corriger » certains paramètres des réponses qu'il reçoit à destination du client afin que la transparence puisse être complète.

Pour certaines opérations[ACV00][AC02], le μproxy peut avoir à contacter plusieurs de ses sous-serveurs et attendre une réponse avant de fournir la réponse au client, ce qui peut être plus ou moins coûteux selon leur fréquence.

5.2.2.4 Cuckoo NFS (*Layered clustering for NFS*)

Ce système récent [KG02] (fin 2002) développé au CMU permet en intercalant un module intelligent à plusieurs clients NFS2 d'accéder de manière uniforme à plusieurs serveurs Cuckoo NFS agrégés. Ceux-ci implantent un sur-ensemble du protocole NFS, les nouveaux appels servant à gérer les extensions que proposent le modèle.

Ce projet offre une réplication des données fréquemment lues et rarement mises à jour sur plusieurs serveurs afin d'améliorer les performances d'accès aux fichiers. En utilisant un système de traces, il permet de remarquer pendant les temps d'oisiveté du

système quels sont les « points chauds » et donc, potentiellement intéressants à répliquer, pour diminuer la charge globale sur un serveur.

Les serveurs stockent les données telles quelles (répertoires et fichiers sont stockés directement) et une table des réplicats permet de gérer les copies et de les invalider lorsque cela s'avère nécessaire.

5.2.3 En modifiant le code client

Certains systèmes choisissent de conserver le code du serveur et préfèrent localiser les modifications à apporter sur le client. Cette approche peut sembler problématique car si l'on se place dans le cadre d'une grosse grappe de machines, du point de vue de l'administrateur, il est souvent préférable de limiter les modifications à un nombre restreint de machines serveurs qui peuvent être contrôlées aisément plutôt que d'installer des nouveaux modules systèmes sur les clients. En effet, si chaque client doit embarquer une version modifiée du système, cela peut poser des problèmes, notamment au niveau de la stabilité du système et/ou du support-vendeur pour la machine.

5.2.3.1 Bigfoot-NFS

Ce système [KMM94] utilise des modifications sur le client qui lui permettent de stocker leurs fichiers sur plusieurs serveurs NFS de stockage. Le code serveur n'étant pas modifiée, les machines serveurs sont utilisées telles quelles.

La hiérarchie des répertoires est répliquée sur tous les serveurs, les fichiers résident sur un seul serveur et deux fichiers ne peuvent pas avoir le même nom. Le mécanisme de `LOOKUP` est implanté par le biais d'une diffusion (*broadcast*) sur tous les serveurs.

Cependant plusieurs limitations sont présentes dans ce modèle, notamment le fait que deux fichiers ne puissent avoir le même nom dans le système.

Une autre limitation de notre point de vue est que le grain de répartition est le fichier, ce qui peut poser problème si l'on souhaite transférer des gros fichiers, un seul serveur va finalement être utilisé. Un autre aspect négatif, à nos yeux, est que le client requiert d'être modifié, ce qui éloigne ce système de nos objectifs.

5.2.3.2 Expand (*Expandable Parallel File System*)

Ce système a été développé de manière à pouvoir fournir une base pour une implantation de MPI-IO [CGC+02]. Les modifications se situent essentiellement sur le client et ne requièrent donc pas de modifications du code de la partie serveur tournant sur les machines stockant les fichiers, ce qui permet donc de pouvoir utiliser des serveurs de différentes architectures pour pouvoir gérer le stockage, la couche NFS/RPC/XDR se chargeant d'en assurer la transparence.

Pour pouvoir être utilisé de façon optimale dans les applications MPI-IO, Expand peut être intégré dans MPI-IO de manière à fournir une implantation optimisée gérant le *striping* des données.

Les fichiers sont actuellement distribués sur l'ensemble des fichiers avec un mode de distribution cyclique configurable. Ces informations de répartitions sont stockées dans un méta-fichier dont le serveur responsable est trouvé en calculant un *hash* sur le nom du fichier. Les méta-données sont stockées comme des « sous-fichiers » sur les serveurs NFS responsable du stockage des morceaux de fichiers.

Si une opération requiert d'accéder à plusieurs serveurs, les requêtes RPC correspondantes sont générées en parallèle par le client qui agrégera alors les réponses de manière à satisfaire la requête de l'utilisateur.

Cette approche bien que performante n'est cependant pas complètement en accord avec les contraintes que nous avons car cela requiert une modification du code faisant fonctionner les clients, ce qui peut s'avérer quelque peu intrusif.

5.2.4 Conclusions

Nous venons donc de présenter plusieurs approches utilisant NFS afin d'améliorer le fonctionnement des serveurs NFS, et ce de manière plus ou moins intrusive pour les clients et/ou les serveurs. Le tableau récapitulatif 5.1 synthétise le fonctionnement des différents projets.

En utilisant le critère de distribution des données, nous nous rendons compte que seul Slice correspondait à peu près à nos critères mais que d'une part, son mode de fonctionnement (filtre réseau sur les clients) et son seul support BSD en limitait l'utilisation que nous pouvions en faire. Ainsi, puisque nous n'avons pas trouvé de système qui satisfasse complètement nos critères, nous avons donc décidé de nous orienter vers la conception d'un tel système.

5.3 Rappels de quelques critères de conception

Pour la conception de NFSP, nous avons retenu plusieurs critères qui nous ont semblé essentiels pour l'installation et l'utilisation dans une grappe usuelle de PC dans le cadre de transferts grappe à grappe :

– Le système doit être simple à installer et désinstaller : le client ne doit pas être modifié, ce qui pose des problèmes notamment au niveau des limitations des protocoles utilisés. En effet, charger du code noyau sur plusieurs centaines de clients peut poser des problèmes sérieux, s'il n'a pas été suffisamment testé. En revanche, conserver le code déjà existant évite d'introduire une nouvelle source de *bugs* et permet, par exemple, de conserver le support d'un vendeur.

Nom	Environnement				Fonctionnalités			Pérennité	
	Intrusivité	Pré-requis	Linux	API	Données	Méta-données	Cohérence	État	Licence
SFS	0	+	*	VFS	0	0	+	+	O
NFS^2	0	+	?	VFS	+	+	+	?	?
Routeur NFS Mirage	0	+	?	VFS	+	0→+	+	+	?
Slice	0→++[1]	++	BSD	VFS	++	0→+	+	+	?
Cuckoo NFS	0	+	?	VFS	+	+	+	+	?
Bigfoot NFS	++	+	0	VFS	+	++	?	-	?
Expand NFS	++	+	?	sp.	++	+→++	?	+	?

TAB. 5.1 – Récapitulatif de divers systèmes utilisant NFS
Les critères ont été définis dans le tableau 2.1 page 30.

– Les grappes que nous utilisons sont des machines « *off-the-shelf* », comme par exemple des machines de bureau comme celles que nous avions à disposition avec le projet ID/HP i-cluster [RAM+01], c'est-à-dire qu'elles n'ont pas de périphériques particuliers si ce n'est une carte réseau et un disque dur (moins performant qu'un disque de serveur). Le réseau d'interconnexion est celui trouvé au sein de nombreuses entreprises ou laboratoires pour les stations de travail, à savoir un réseau Ethernet 100 commuté.
– Comme nous l'avons déjà indiqué, les données doivent être distribuées afin que les vitesses de transfert agrégées d'une grappe à une autre puissent être cumulées pour remplir les liens gigabits (ou plus) avec du matériel tout à fait standard, sans avoir recours à de nombreux règlages qui peuvent s'avérer dans la pratiques, longs et délicats.
– Cela peut être vu comme une conséquence du point précédent, mais agréger les espaces de stockages inutilisés des nœuds de calcul est aussi un aspect intéressant puisqu'autrement, cet espace est inutilisé donc « perdu ».

De plus, comme l'utilisation de serveur NFS est bien connue des administrateurs, elle permet de procéder à une installation et à une utilisation rapide, sans remettre en cause l'installation des machines d'une grappe. Les versions NFS ont aussi été développées comme des modifications et des ajouts à des codes existants tout en essayant de minimiser les modifications de ces codes afin, d'une part de limiter l'introduction de *bugs*, et d'autre part, de permettre de laisser les chemins de codes standards pour utiliser la version « normale » du serveur.

5.4 Principes de NFSP et applications envisagées

Afin de développer un système de fichiers pouvant satisfaire les pré-requis que nous nous étions fixés nous avons souhaité faire évoluer un serveur NFS en système distribué, c'est-à-dire un NFS parallèle. Différents systèmes ont été développés, par le passé, en ce qui concerne la distribution des données et des méta-données, mais peu ont été pensés pour avoir un système de transfert haut-débit entre deux grappes (pour cela il faut plutôt regarder du côté des coûteux systèmes de stockage dédiés à la HPSS [HGFW02] ou bien DPSS [Lab] qui disposent de systèmes de transferts parallèles optimisés).

En gardant à l'esprit, d'une part les points développés dans la section précédente, et en remarquant d'autre part, l'omniprésence de NFS dans le monde UNIX, et ce, bien que les défauts en soient connus (cohérence temporelle, verrous, etc…), nous avons donc choisi de développer diverses solutions visant à améliorer l'utilisation qui peut être faite de ce protocole dans le cadre d'une architecture distribuée.

En partant de l'architecture usuelle d'un gros serveur NFS présentée dans la figure 5.4 page 80, et souhaitant avoir une architecture répartie de manière transparente nous avons été amenés à adapter cette architecture à notre architecture matérielle à savoir l'*i-cluster*.

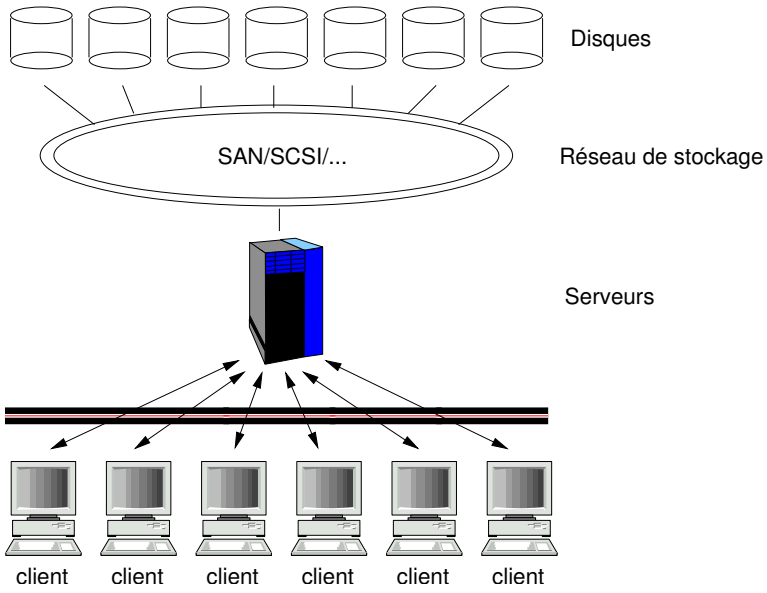

FIG. 5.4 – Serveur NFS classique

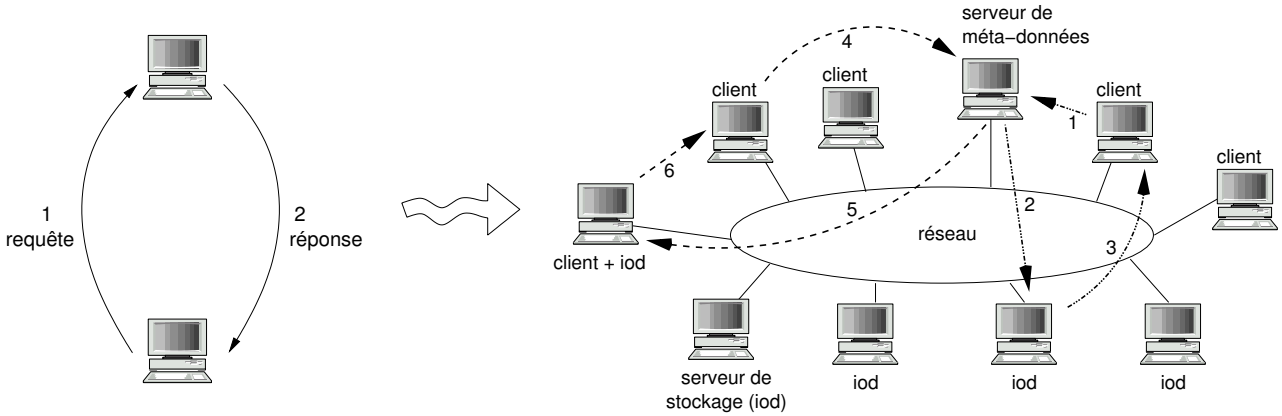

FIG. 5.5 – Passer d'un mode NFS à un mode NFSP

Des moyens ont ensuite été développés afin de pouvoir tirer partie de la répartition des données pour effectuer les transferts de grappes à grappes, puisque nous disposions d'une brique de base pour ceux-ci. Nous nous sommes principalement concentrés sur la version 2 du protocole car elle disposait d'une implantation en mode utilisateur aisément disponible (les administrateurs n'ayant pas à modifier leur noyau). La version 3 a commencé à être étudiée mais certains problèmes demeurent, aussi les présenterons-nous ultérieurement.

La version 4 a subi de profondes mutations (notamment le passage à un mode connecté) qui risquent de poser des problèmes si l'on souhaite effectuer une répartition de la charge. De plus, le protocole est relativement récent, ce qui fait que les codes clients et serveurs sont eux aussi « jeunes » et donc relativement peu testés, ce qui a pour conséquence de ne pas favoriser un remplacement rapide des anciennes versions.

Pour résumer, le but principal de NFSP est d'offrir une implantation distribuée d'un serveur NFS afin d'agréger l'espace inutilisé des disques d'un *cluster* pour offrir une amélioration des performances dans un cadre local et dans un cadre d'échanges inter-LAN comme brique de base pour du transfert à haut-débit. Un autre type d'applications, qui peut aussi tirer partie de NFSP concerne plusieurs applications à vocation scientifique, telles que certains programmes de physique ou bien de biologie. En effet, certains modèles d'accès de ces applications sont particulièrement bien adaptés à un fonctionnement avec le modèle que nous envisageons puisque de nombreux noeuds lisent d'abord des données de manière parallèle (par exemple une séquence génétique) pour ensuite effectuer les opérations qui les intéressent. Les résultats sont ensuites écrits, mais les données à envoyer sur le stockage physique ont une taille d'un autre ordre de grandeur que les données lues.

De plus, nous avons souhaité conserver une contrainte forte, à savoir un respect du protocole NFS de façon à ne rien avoir à installer de nouveau sur les postes clients. Afin d'avoir des performances, il faut éviter de surcharger la bande passante du serveur principal en faisant répondre les entités de stockage directement. Ainsi, puisque le temps passé sur le routage de la requête vers l'entité de stockage est très inférieur au temps passé sur l'entité de stockage pour satisfaire la requête, de meilleures performances peuvent être observées lorsque plusieurs clients accèdent à des fichiers en parallèle. En revanche, puisqu'il ne reste toujours qu'un seul serveur vu par le client (par respect du protocole NFS), les clients ne verront probablement pas de gain en écriture, si ce n'est par le fait que les E/S disque ne seront pas réalisées sur le serveur mais sur plusieurs serveurs d'E/S (les E/S réseau sont moins coûteuses que les E/S disque), ce qui permettra d'alléger un peu la charge du système.

Pour fournir quelques points de repères numériques, notamment en ce qui concerne les ordres de grandeurs évoqués, nous pouvons observer les performances de disques et de réseaux actuels. Par exemple, les disques durs standards (IDE 7200/rpm sur l'i-cluster) ont une latence d'accès moyenne de 9ms, c'est-à-dire qu'avant de pouvoir accéder à la donnée le disque met cette durée avant de pouvoir la fournir (mouvement des pièces physiques). Si l'on regarde l'évolution de ces temps depuis quelques années, ceux-ci ont, certes, diminué mais de manière relativement faible ; des facteurs de

cette diminution étant entre autres, l'accélération de la rotation des disques et l'augmentation de leur densité de stockage. Si l'on compare maintenant ces temps à ceux de simples cartes Ethernet 100, ces latences sont au moins 50 fois plus faibles : pour fixer les idées, entre deux nœuds de l'i-cluster, la latence d'un ping ICMP, c'est-à-dire avec les surcoûts de traitement liés à l'OS, est seulement d'environ 0.150ms... D'autre part, certaines architectures, certes dédiées telles que Gigabit Ethernet, Myrinet ou SCI, permettent même d'avoir des latences encore plus faibles, mais cela se paie par un prix de matériel d'inter-connexion beaucoup plus élevé.

Il faudra donc mettre en place des serveurs de données (nous les appellerons iod) ainsi qu'un protocole de communication entre le méta-serveur (appelé par la suite nfspd) et ces iods. Nous décrirons, par la suite, les divers choix que nous avons effectués pour les implantations.

5.5 Conclusion

Nous avons présenté dans ce chapitre plusieurs solutions utilisant le protocole NFS dans divers rôles ayant pour la plupart pour but d'améliorer l'extensibilité du stockage. Cependant, nous sommes arrivés à la conclusion qu'aucune ne proposait de solution satisfaisante à nos problèmes.

Nous avons ensuite rappelé les buts que nous recherchions et qui nous ont poussé à développer un système de fichiers distribué, NFSP, afin de les résoudre.

Nous avons, par la suite, exposé les modifications à mettre en place et à apporter à un serveur standard pour qu'il puisse devenir une implantation distribuée du protocole NFS que nous appelons NFSP.

Ces principes étant énoncés de manière succincte, nous allons maintenant décrire l'implantation des différents prototypes que nous avons été amenés à réaliser afin de pouvoir étudier leur fonctionnement.

Chapitre 6

Implantations des prototypes

Ce chapitre vise à présenter les implantations réalisées de notre proposition. Nous commencerons par présenter de manière un peu plus détaillée le fonctionnement de NFS puis nous aborderons les problèmes rencontrés lors de la réalisation des prototypes.

6.1 Un peu plus de détails sur NFS

Comme nous l'avons écrit dans la section 4.1.1 page 44, NFS est un protocole en couches s'appuyant sur un fonctionnement par RPC, qui eux mêmes utilisent un système d'encodage des données par XDR. Ce chapitre va principalement concerner les versions 2 et 3 du protocole.

6.1.1 Format des données : XDR

Le format XDR[Sun87, Sri95b] (*External Data Representation*) définit un moyen d'échanger des données évoluées de manière normalisée entre deux machines d'architectures é-ven-tuel-lement différentes.

Sont définis des types de base (plusieurs types et tailles d'entiers, des énumérations, des flottants, des zones de données opaques de taille fixe ou variable, des chaînes de caractères) ainsi que des types plus complexes tels que : structures, unions, tableaux (taille fixe ou variable), etc.

Usuellement les types sont définis dans un fichier d'interface qui est ensuite transformé par une application (`rpcgen` pour le C par exemple) en fonctions de sérialisation et de dé-sérialisation des structures, ces dernières pouvant ainsi être envoyées sur le réseau sans se soucier de l'ordre des octets (*little/big endian*).

L'émetteur manipule les données dans un format natif puis les confie à la fonction de sérialisation, le récepteur va alors appeler la fonction de dé-sérialisation qui reconvertira les données venant du « fil » vers son propre format natif. Bien sûr, ce mode

de fonctionnement ne permet pas de transmettre des références et oblige simplement deux machines d'architectures quelconques à respecter un format d'échange commun.

6.1.2 Des appels à distance : RPC

Les RPC[Sun88, Sri95a] (*Remote Procedure Call*) permettent comme leur nom l'indique d'exécuter des procédures à distance. Le mode de fonctionnement de ces appels est un mode client-serveur classique.

Le client et le serveur partagent usuellement une définition des procédures (en langage RPC) et des structures communes. Afin d'être traitable, l'appel à une fonction RPC utilise trois éléments discriminants :

1. le numéro du programme : ce numéro définit le type de service fourni. L'attribution de ces numéros est arbitraire et respecte quelques règles d'une manière assimilable à ce que se passe avec les noms de services Internet bien connus et leurs ports. Ainsi, les services comme NFS ou NIS se voient attribuer un numéro qui permet aux clients intéressés de les contacter.
2. la version du programme : ceci permet d'offrir plusieurs versions d'un même service. Par exemple, la plupart des serveurs NFS offrant le mode NFS3 continuent d'offrir le mode d'accès par NFS2.
3. le numéro de la procédure : les procédures sont simplement numérotées.

Un serveur RPC spécial (`portmap`) assure un service d'annuaire des services disponibles. Lorsqu'un client recherche un serveur RPC, il s'adresse en général d'abord au serveur RPC `portmapper` (qui est assuré par le démon `portmap`) en fournissant le numéro de programme et de version recherchés, et il obtient en cas de réponse positive le numéro de port (TCP ou UDP) à utiliser pour contacter le serveur RPC demandé.

La couche RPC permet aussi de gérer la fiabilisation des transmissions par le biais de ré-émissions / *timeout*, ce qui s'avère important dans le cas de l'utilisation sur des couches de transport non-fiables telles que celles fournies par les datagrammes UDP.

Le mode de fonctionnement des RPC ici décrit est intrinsèquement synchrone car tous les appels doivent recevoir une réponse, ce qui peut poser des problèmes de performance. L'utilisation de techniques de programmation *multi-thread* peut permettre de limiter ces problèmes en récupérant les temps de communications et en les allouant à du traitement.

Cependant, puisque RPC définit principalement un système de passage de messages, il peut aussi être utilisé pour effectuer des opérations par lot (*batch*) en envoyant plusieurs requêtes sans attendre de retour. Dans ce cas-là, un canal de transmission fiabilisé tel que celui offert par TCP doit plutôt être utilisé.

Une autre utilisation peut aussi consister à l'utiliser comme un protocole de diffusion (*broadcast* ou *multicast*), auquel cas, l'utilisation de datagrammes UDP est plus naturelle. Ainsi, une des fonctions du `portmapper` est de permettre de jouer un rôle de

mandataire. Prenons l'exemple d'un client qui souhaite effectuer une diffusion RPC sur un réseau local c'est-à-dire une demande RPC à une même fonction sur toutes les machines du réseau. Une solution est de diffuser une requête par *broadcast* UDP sur le réseau à destination des `portmapper` (ceux-ci ont un port fixe). Ces derniers vont alors effectuer l'appel local si le service et la procédure demandée ont été enregistrés. Usuellement, seules les réponses positives reviennent aux clients (mais cela peut varier en fonction de l'application).

6.1.3 Fonctionnement d'un client NFS

La partie client de NFS est intégrée au système d'exploitation et assure la conversion des opérations usuelles effectuées au niveau de la VFS en des requêtes NFS permettant de les satisfaire. Sous Linux un *thread* kernel (`rpciod`) permet de régler la gestion des requêtes RPC avec les serveurs. Un cache de données effectué en mémoire permet d'éviter les accès trop fréquents au serveur.

Selon les options données au client (lors du montage de la partition), ce dernier peut exhiber plusieurs comportements. Ainsi, sur un client Linux peuvent, par exemple, être définis les paramètres suivants :

- taille maximale des lectures (`rsize` - défaut 1024B) / des écritures (`wsize` - défaut 1024B)
- paramètres de retransmission : *timeout* de base (`timeo` - défaut 0.7s - croît en doublant jusqu'à 60s et cause alors un *timeout* majeur ; le comportement alors suivi dépendra du type de montage `hard/soft`)
- type de montage : en `hard`, le client ré-essaie tant qu'aucune réponse n'a été reçue, en montage `soft`, celui-ci renvoie une erreur d'E/S (`EIO`) à l'application en cas de *timeout* majeur
- interruption possible : le paramètre `intr` par défaut désactivé permet de renvoyer à un client `hard` une erreur `EINTR`, ce qui permet de pouvoir arrêter l'application « plus proprement » en cas de défaillance du serveur
- de nombreux autres paramètres sont disponibles mais moins cruciaux et pour la liste exhaustive, le lecteur pourra se reporter à la page *man* `nfs(5)`

Les données peuvent être cachées 3 secondes en mémoire par défaut mais certaines extensions (non standard) peuvent permettre d'effectuer des E/S sans cacher les données ni les méta-données, ce qui peut avoir un intérêt pour certains types d'applications mais risque de se répercuter par une charge du serveur plus élevée.

6.1.4 Le serveur NFS Linux en espace utilisateur

Afin de tester la validité de l'approche que nous envisagions, nous avons commencé par utiliser le démon NFS Linux en mode utilisateur.

Certes, ce choix peut sembler poser quelques problèmes notamment au niveau des performances amoindries en raison des surcoûts liés à l'utilisation de processus en

mode utilisateur, en particulier le nombre plus élevé de changements de contextes et le surcoût occasionné par le nombre de recopies mémoires entre les espaces utilisateur et noyau.

Cependant, cette perte de performances offre cependant un confort accru au niveau de la programmation, de l'utilisation et du *débuggage*, ce qui fait qu'il peut être utilisé Le serveur a aussi été développé comme un serveur simple mono-*threadé*, ce qui contribue à en limiter aussi les performances (mais il arrive cependant sans trop de problèmes à remplir un réseau à 100Mb/s). De plus, une autre limitation est que seule la version 2 du protocole NFS est supportée, ce qui peut par exemple poser certains problèmes dans le cas des accès à des fichiers de plus de 2GB.

Un autre aspect important à considérer est la faible intrusivité de cette application : puisqu'elle tourne complètement en mode utilisateur, elle peut être installée sans charger de nouveaux modules noyaux, ni *patcher* à bas niveau un système existant.

En plus du démon `nfsd`, un serveur `mountd` est livré avec le système et permet de gérer cette partie du protocole (export des partitions). Le serveur NFS et le démon de montage partagent tout deux une connaissance du calcul des *handles* afin que le démon de montage puisse fournir aux clients un *handle* initial ayant un sens pour le serveur.

6.1.5 Le serveur NFS Linux en espace noyau

Ce serveur est l'implantation disponible dans les noyaux Linux depuis maintenant plusieurs années. Son architecture est *multi-threadée* et plusieurs *threads* peuvent répondre aux requêtes afin d'augmenter sa capacité de traitement.

Ce service est disponible sous la forme d'un module noyau (ou bien intégré en « dur ») et requiert l'utilisation de programmes annexes en mode utilisateur qui sont disponibles dans le paquet `nfs-utils` afin de gérer sa configuration. Parmi les outils fournis, les trois utilitaires principaux sont :

- `rpc.nfsd` : Cet utilitaire permet de « lancer » le service NFS en précisant par exemple le nombre de *threads* qui doivent être consacrés au fonctionnement du système. Ce paramètre est à adapter en fonction de la charge du serveur (beaucoup de *threads*), du nombre de clients, de la puissance du serveur, ... En effet, la gestion d'une requête étant synchrone au sein d'un *thread*, il vaut mieux avoir plus de *threads* quitte à ce qu'ils soient inutilisés pour pouvoir absorber les requêtes des clients, sachant qu'avoir trop de *threads* inutilisés risque de dégrader les performances du système (mais moins que de ne pas en avoir assez !)
- `rpc.mountd` : Comme son nom l'indique, il permet de servir les demandes de montage émanant des clients. De même que son équivalent dans le prototype en mode utilisateur, ce service RPC exporte un handle initial pour les clients. Pour éviter la duplication de code, il obtient ses paramètres initiaux (par exemple *handles* des exports) par le biais d'un appel système spécifique (`nfsservctl()`)

– `exportfs` : Cet utilitaire permet de modifier sans redémarrer le serveur la liste des exports mis en place et utilise lui aussi l'appel système `nfsservctl()` afin de communiquer avec le serveur.

De par son intégration dans le noyau et son implantation *multi-threadée*, les performances sont bien évidemment meilleures que la version du démon utilisateur (minimisation des recopies entre espace utilisateur et noyau) et il a aussi accès aux structures internes du noyau qui peuvent permettre d'optimiser les performances.

6.2 Nature des modifications à mettre en œuvre

Nous avons souhaité conserver la gestion de la majorité des opérations sur les méta-données par le code des serveurs actuels. Ainsi la plupart des opérations ne requérant pas d'accéder aux données sont traitées sur le méta-serveur.

Puisque le système de fichier offre nativement la plupart de la gestion de ces opérations et offre un cache cohérent nous avons souhaité conservé la correspondance un fichier NFSP = un fichier de méta-données sur le méta-serveur (= méta-fichier).

Le méta-serveur devra donc, lors d'un accès, ouvrir le méta-fichier et récupérer les informations nécessaires à la résolution de la requête. La version 2 du protocole requérant d'avoir les attributs du fichier dans la réponse pour certaines requêtes, il faudra donc les transmettre aux serveurs de méta-données. Ceux-ci devront alors avoir toutes les informations pour pouvoir générer les réponses à destination des clients (IP, port, numéro xid de requête RPC, ...)

D'autre part, il y a aussi un « mauvais » cas d'accès : si une requête demande des données qui se situent sur deux serveurs de données, il faudra alors fournir au premier iod de quoi faire suivre la requête au deuxième serveur, ce dernier pourra alors compléter la requête et la renvoyer au client.

6.3 Premier prototype : mode utilisateur

Pour réaliser la première version de NFSP, nous avons choisi d'utiliser le serveur NFS en mode utilisateur afin d'étudier son comportement lorsqu'il était stressé et valider le fonctionnement du principe que nous avions retenu pour NFSP. Les articles [LD02a] et [LD02b] décrivent l'implantation de manière concise.

6.3.1 Implantation

La première version a été réalisée sous formes de modifications apportées au serveur NFS en mode utilisateur. Celui-ci devenant alors un interpréteur de requêtes NFS, peut

en fonction du type de requêtes transmettre la demande à l'entité de stockage qui est susceptible de la traiter.

Au niveau des entités de stockage, ce sont des entités qui reçoivent des requêtes depuis un méta-serveur et qui renvoient des réponses au client en se faisant passer pour le serveur, ce qui nous a orienté de suite sur des solutions de *spoofing*.

Afin de ne pas avoir à modifier les piles réseaux (gérer des *offload* TCP aurait requis de toucher à du code de la pile réseau), nous nous sommes concentrés sur le fonctionnement en UDP qui est actuellement le protocole le plus souvent utilisé pour gérer des serveurs NFS2 et NFS3.

6.3.1.1 Les méta-données

Afin de limiter les modifications à réaliser sur le serveur, nous avons souhaité continuer à utiliser le système de fichiers sur lesquels les fichiers étaient stockés et ainsi pouvoir continuer à offrir une vue de système de fichiers classique, c'est-à-dire qui permette de gérer des liens « mous » et durs. Ainsi un fichier créé sur un export NFSP, sera en fait stocké comme :

- un fichier sur le méta-serveur (= un méta-fichier) : c'est un fichier enrichi de diverses informations supplémentaires qui sont nécessaires au bon fonctionnement de NFSP telles que, par exemple, un *cookie* (dont nous préciserons l'utilité par la suite) et la taille réelle du fichier que le client doit voir. Bien sûr, d'autres informations peuvent être ajoutées à ces méta-informations pour gérer la localisation des blocs de données ou bien des versions de blocs de données, ... Cependant, on doit conserver en mémoire le fait que, pour chaque requête, ces fichiers peuvent avoir à être consultés et que par conséquent, leur structure et les opérations à effectuer pour gérer la distribution des données doivent souvent être simples (par exemple, une simple opération mathématique pour savoir sur quel `iod` se trouve tel bloc en fonction de son *offset*).
- des fichiers sur les iods : ceux-ci portent un nom qui est encodé à partir des propriétés du méta-fichier. Nous présenterons plus en détail le format de ces données dans la section 6.3.1.7 page 95.

D'autre part, nous avons choisi de laisser gérer une partie des méta-données par le système de fichier sur lequel étaient stockées les fichiers de méta-données. Typiquement, nous avons laissé le système gérer les permissions et les droits sur les fichiers ainsi que la gestion des diverses dates reliées au fichier[1].

De plus puisque nous souhaitons conserver le fonctionnement des liens « durs », nous allons considérer qu'un fichier correspond à un inode unique. Ainsi, c'est grâce au système sous-jacent que cette fonctionnalité pourra être conservée car lorsque le client

[1] À noter que la gestion de la date de dernier accès requise par la spécification POSIX n'est pas gérée mais à notre défense, on peut remarquer que de nombreux serveurs de fichiers ou bien systèmes locaux de fichiers ne la prennent pas en compte car pour chaque accès en lecture ou en écriture, une écriture dans les méta-données du fichier s'avère nécessaire pour garder trace du dernier accès !

demandera un lien dur sur un fichier, le système créera aussi un lien dur sur un méta-fichier, ce qui aura pour conséquence de produire l'effet escompté.

Par la suite, les données correspondant à un fichier seront situées sur un iod qui pourra être retrouvé à partant de l'inode du méta-fichier et de sa graine[2] (ou *cookie*). D'autres paramètres auraient pu être utilisés tels que :

– un *hash* sur le nom du fichier : mais dans ce cas là, la gestion des déplacement de fichiers ou de re-nommage s'avère beaucoup plus complexe. Bien sûr, un système de redirection pourrait être utilisé mais cela ne nous semble pas opportun car à chaque redirection à suivre, la latence du service est augmentée, ce qui risque de provoquer une expiration du *timeout* des clients.
– une gestion des méta-données « virtuelles » : celles-ci peuvent être stockées dans une base de données par exemple mais certaines opérations usuellement assurées par le système de fichier sous-jacent peuvent alors s'avérer très difficiles à implanter correctement (liens durs, etc). De plus, comme le contenu des méta-données est fréquemment utilisé, il vaut mieux qu'il puisse bénéficier du cache offert par le système de fichiers local.

Ainsi, grâce à cette architecture, la plupart des opérations sur les méta-données et donc sur les méta-fichiers sera résolue directement sur le méta-serveur. Par exemple, la création de répertoires, les déplacements de fichiers sur la partition exportée seront gérés de cette manière sans besoin d'implantation particulier.

De plus, la gestion des verrous simples (`flock()`) sera assurée sur les méta-fichiers qui seront simplement vus comme des fichiers à verrouiller. En revanche, pour les verrous plus évolués (zones de fichiers), cette approche ne fonctionnera pas sans support supplémentaire.

Cependant cette approche a un point négatif qui est de ne pas pouvoir déplacer « facilement » un répertoire de méta-données car actuellement le système de nommage des fichiers de données est fonction du numéro d'inode du méta-fichier, ce qui peut poser des problèmes si l'on souhaite effectuer une sauvegarde des méta-fichiers. Ce désavantage est compensé par le fait que les renommages de fichiers (sans changement d'inode) s'effectuent sans avoir à renommer les fichiers contenant les données sur tous les serveurs. Il est toutefois possible de sauvegarder un système NFSP mais il faudra alors recourir à un utilitaire pour la réaliser et restaurer les bons noms des données.

6.3.1.2 Rôle du *cookie* et déroulement d'un effacement de fichier

Si l'on ne repérait les fichiers et leur données que par l'inode, un problème se produirait dans le cas où l'on effectuerait un effacement, puis une re-création d'un autre fichier. En effet, puisque nous souhaitions disposer d'une gestion de l'effacement asynchrone, il fallait que l'on puisse *logger* la demande d'effacement et effacer le méta-fichier afin que les données ne soient plus accessibles, ce qui laissait une fenêtre où l'on pouvait encore accéder aux données, malgré l'effacement.

[2]Un numéro choisi aléatoirement afin de rajouter un paramètre discriminant sur les fichiers.

Ainsi, pour ne pas avoir à payer le coût d'une synchronisation avec les iods, il a été décidé de procéder à un effacement asynchrone se déroulant selon les étapes suivantes :

1. le client envoie sa demande d'effacement au méta-serveur
2. le méta-serveur note dans une queue que le fichier doit être effacé
3. le méta-fichier est effacé
4. l'acquittement est renvoyé au client
5. lorsqu'il y a suffisamment de demandes d'effacement ou qu'un *timeout* expire (inactivité du système), une demande de récupération des données est envoyée aux iods

Si l'on avait opté pour représenter, par exemple, un effacement par un renommage dans un répertoire différent suivi d'un effacement, cela aurait pu poser des problèmes, notamment en laissant le fichier accessible bien qu'il ait été officiellement effacé en réutilisant le même *handle*.

Le *cookie*, permet d'avoir de fortes chances que, même si l'inode du fichier effacé est réalloué, le client ne puisse pas accéder aux anciennes données, car son nouveau *cookie* sera différent.

6.3.1.3 Solutions de *spoofing* UDP

Afin que les iods puissent répondre directement aux clients et ainsi éviter de saturer la bande passante du méta-serveur, une solution consistant en la mise en place de techniques de *spoofing* est vite apparue.

Pour rappel, le *spoofing* consiste à créer un paquet (UDP dans notre cas) pour lequel les entêtes ont été modifiés pour sembler venir d'une source physique différente de celle qui les a générés. De telles techniques sont souvent utilisées à mauvais escient dans les attaques de dénis de service distribués[CER96] ou bien dans les tentatives de *scans* de réseau afin de noyer l'adresse de la machine qui scanne parmi un grand nombre d'adresses avec des adresses leurres[3].

Dans le cas qui nous intéresse nous allons l'utiliser pour masquer le fait que les paquets viennent d'un serveur de stockage et non pas du méta-serveur comme le client pourrait s'y attendre. Le fonctionnement général avec la place du *spoofing* est illustré sur la figure 6.1 page suivante.

Dans le code du `iod`, une pile de *spoofing* UDP a été implantée pour pouvoir gérer ces émissions. Les premiers tests utilisant uniquement *spoofing* au niveau IP ont montré (certes au bout d'un moment) qu'il y avait un problème dans cette gestion avec la configuration des *switchs* Ethernet.

[3]Par exemple, avec l'un de ces outils les plus connus, `nmap` [Fyo], cela peut être fait avec l'option `-Ddecoy_host1,...`

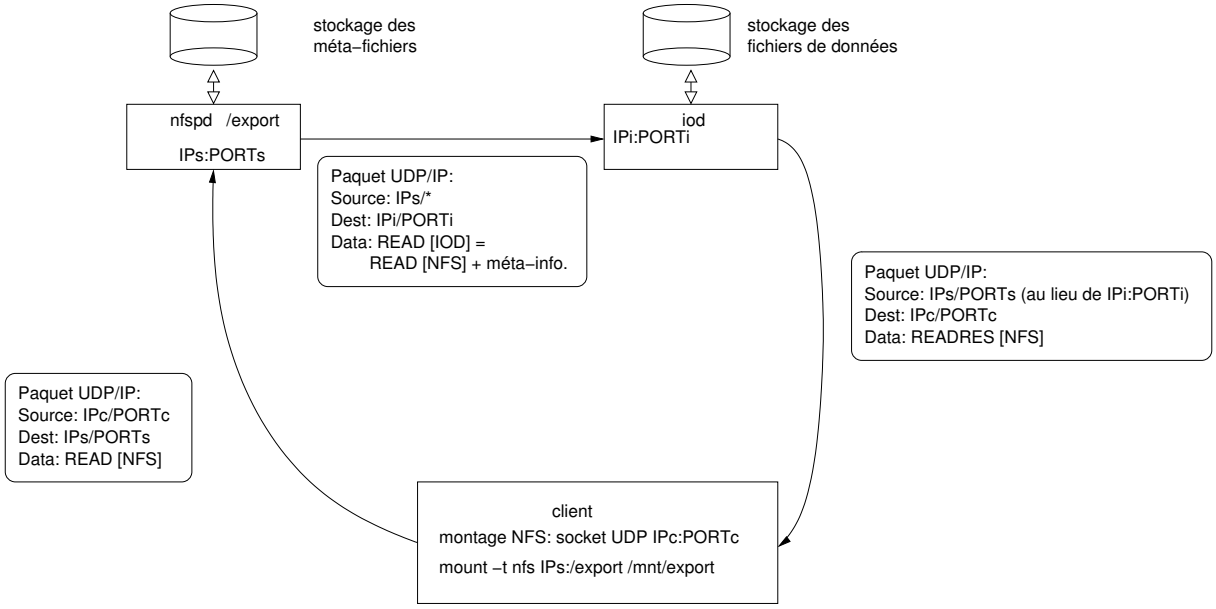

FIG. 6.1 – Utilisation du *spoofing* dans NFSP

En effet, puisque l'on écrivait simplement la destination des paquets avec une source IP imitée, les paquets ne savaient pas où être routés et finissaient par aller sur le routeur du *cluster* où grâce/à cause d'un problème de configuration ils étaient réinjectés dans la grappe au lieu d'être rejetés[4] car ils portaient l'adresse MAC du routeur au lieu de la machine de destination.

Nous avons donc, par la suite, rajouté aux informations transmises par le méta-serveur à l'iod, l'adresse MAC du client afin de pouvoir effectivement gérer ce mode d'émission propre, mais nous avons alors dû utiliser un *socket* RAW à un niveau inférieur (*link layer socket*) dans lequel nous composions nous-même les entêtes Ethernet du paquet (adresses MAC).

Si l'on avait souhaité mettre en place une gestion du NFS avec RPC sur TCP, les problèmes auraient été beaucoup plus nombreux car il aurait fallu que les réponses puissent s'intercaler dans le flux légitime qui existait entre le serveur NFS et son client, et donc il aurait par exemple fallu ajouter en plus des modifications noyau, une synchronisation nécessaire entre l'`iod` et le `nfspd` afin de simplement pouvoir gérer les numéros de séquences TCP. Une approche similaire à celle développée dans [SHHP00] (optimisation des *forwarders* de flux TCP) n'aurait pas permis de pouvoir éviter de repasser par le méta-serveur, qui aurait alors dû effectuer l'entrelacement des données et tous les gains espérés auraient été perdus.

Des techniques d'*offload* TCP[ASDZ00] (les données arrivent sur le méta-serveur avec les iods répondent) auraient, à notre avis, requis un matériel spécialisé (réseau très

[4]Les paquets UDP suivaient le trajet : iod -> *switch* -> routeur -> *switch* -> client au lieu du chemin attendu iod -> *switch* -> client.

faible latence) comme ce que l'on peut voir dans le service de caches web sur mémoire distribuée partagée *Whoops !*[Cec02] utilisant un réseau SCI . Néanmoins, dans le cas le plus simple, le méta-serveur va devoir ignorer les paquets TCP venant de la connexion du client pendant que les iods génèrent les requêtes et ne pas envoyer de réponses aux clients avant d'avoir reçu un message de déblocage de l'iod. Une telle approche n'a à notre avis rien de peu intrusif car elle est doit être intégrée finement au sein de la pile réseau. De plus elle requerrait une interaction plus poussée de gestion de la cohérence des connexions entre le méta-serveur et ses iods (qui peuvent changer, à priori, à chaque accès) dans notre cas, ce qui quelque part revient à réduire les bénéfices d'un système d'*offload*.

6.3.1.4 Mise en place de « crochets » sur le serveur

Les modifications ont principalement consisté à mettre en place les fonctions adéquates qui permettraient de gérer les méta-données.

Des *wrappers* autour des fonctions usuellement utilisés `stat()`/`lstat()`/`fstat()` ont été réalisées afin d'encapsuler l'exécution de ces fonctions par des opérations d'entrées/sorties sur le contenu des méta-données et ce afin de pouvoir laisser une bonne partie du code du serveur non-modifiée.

CREATE Lorsqu'un fichier est créé sur une partition NFS normale, le serveur crée sur le système de fichier local le fichier par les appels standards (`open()` avec option `O_CREAT` par exemple).

Dans le cadre d'un export NFSP, la création d'un fichier requiert quelques précautions car on doit en effet créer un fichier et remplir son contenu avec des méta-informations que le système ne verra pas. Ainsi nous avons ajouté la taille réelle du fichier, c'est-à-dire celle qui est renvoyée par le système lorsque l'utilisateur demande la taille d'un fichier, et un numéro de graine (*seed*) qui joue le rôle d'un *cookie* et permet de rajouter un paramètre discriminant sur un fichier (en plus de son inode).

Le rôle du *cookie* est de s'assurer que lors d'un effacement d'un fichier suivi d'une re-création, si le même numéro d'inode est attribué au fichier, les données qui sont effacées sur les iods de manière asynchrone ne seront pas accessibles depuis le nouveau fichier.

GETATTR Cet appel permet de récupérer les méta-données du fichier et correspond typiquement avec une application effectuant une des variantes de l'appel système `stat()`. Le client va alors recevoir une structure contenant les différentes informations.

Puisque c'est une fonction n'affectant pas les données elle est entièrement gérée par le méta-serveur et c'est lui qui va répondre au client sans avoir à passer par un quelconque iod. Le serveur lit le contenu du méta-fichier stocké (si c'est un fichier régulier)

et effectue un `stat()` sur le méta-fichier. Il complète alors la réponse avec les informations que le `stat()` du méta-fichier a donné et remplace par exemple la taille par la taille réelle du fichier stocké.

SETATTR Cet appel NFS permet de régler divers paramètres sur un fichier tels que les permissions et les dates d'accès. Il permet aussi de tronquer un fichier mais cette opération est relativement délicate à implanter.

En effet, si un fichier est tronqué, puis qu'il est par la suite agrandi par delà le point de tronquage, les données situées entre la coupure et la nouvelle fin du fichier doivent être lues comme étant des zéros.

Une telle opération a alors un fonctionnement très synchrone. Ce comportement est alors difficile à garantir dans le cadre d'architectures distribuées et l'est encore plus particulièrement dans le cadre de NFS où les clients peuvent cacher les données et les méta-données pendant un certain temps.

Actuellement, nous n'avons pas implanté la bonne gestion de ce tronquage de fichier car la complexité de l'implantation à réaliser a de fortes chances de ne pas servir à grand chose dans le cas où les clients cacheraient les données. Cependant, une possibilité de réaliser cette opération de manière correcte serait de faire une diffusion synchrone (avec acquittement) à tous les iods de la demande de tronquage, ce qui risque de paralyser momentanément le système. Cela peut être éventuellement limité au fichier devant être tronqué mais cela requerrait aussi de garder un état sur le serveur, ce qui rajoute de la complexité au système et donc risque d'augmenter sa charge.

READ La réalisation du READ requiert d'abord de récupérer les méta-informations du fichier. À partir de ces informations et de l'*offset* de la demande, le méta-serveur trouve le serveur d'E/S qui devra servir la requête. Il transmet alors la demande du client, ainsi que les informations nécessaires pour que l'iod puisse générer le paquet adéquat de retour au « bon » client. Celui-ci devra répondre directement au client sans repasser par le serveur, ce qui évitera de gaspiller la bande passante du serveur.

WRITE Le WRITE se déroule d'une manière très similaire au READ. La différence principale étant que les données à écrire doivent elles aussi être transmises à l'iod, ce qui sature la bande passante de sortie du méta-serveur. Cependant, il n'y a pas vraiment d'autre moyen si l'on souhaite conserver une transparence totale vis-à-vis du client, car celui-ci ne connaît que son point de montage, c'est-à-dire le méta-serveur. Cette approche limite les performances en écriture car toutes celles-ci doivent passer par ce point de centralisation. Nous présenterons ultérieurement plusieurs solutions que nous avons envisagées, afin d'augmenter la bande passante disponible en écriture.

6.3.1.5 Cas un peu plus spéciaux

Lorsqu'une lecture est demandée à un endroit qui n'a pas encore été écrit mais simplement alloué (fichier creux ou *sparse files*), le système doit retourner un contenu vide (des zéros) au processus qui lit les données.

Pour gérer l'effacement des données nous avons mis en place un deuxième réseau avec les iods. Les demandes d'effacement sont gérées de manière asynchrone sur le serveur : la demande d'effacement utilisant l'inode et la graine du fichier est mise dans une queue maintenue sur le serveur. Une fois la taille maximale de la queue atteinte ou bien si rien ne s'est produit pendant un laps de temps, la demande d'effacement est envoyée aux iods qui doivent dès lors effacer les données correspondant au méta-fichier effacé.

Il y a pendant ce laps de temps entre l'effacement du méta-fichier et l'effacement des données une fenêtre pendant laquelle si une des machines « plante », les données peuvent ne pas être libérées bien que n'étant plus accessibles. Un moyen pour résoudre ce problème peut être envisagé par la mise en place d'une espèce de journalisation des opérations d'effacement.

Actuellement, un outil annexe permet d'effectuer une opération similaire à celle que ferait `fsck` sur un système local, c'est-à-dire vérifier que tous les données allouées peuvent être accédées. Cependant, cette opération nécessite de mettre le système hors ligne (ou au moins de faire un gel du méta-serveur - c'est-à-dire qu'il ignore volontairement les requêtes des clients, ceux-ci allant ré-émettre jusqu'à être finalement satisfaits lorsque le système sera dégelé).

6.3.1.6 Description du « protocole » nfspd-iod

Le protocole nfspd-iod est en fait limité à effectuer le transfert des requêtes entre le nfspd et les iods. Puisque les requêtes qui arrivent sur le serveur respectent un protocole sans état, il nous a semblé logique de mettre en place un tel protocole, s'appuyant lui aussi sur la couche de transport UDP. Un des avantages à gérer ces échanges de cette manière est qu'il est possible dans le cas où nous souhaitons mettre en place plusieurs frontaux méta-serveurs d'avoir des serveurs de stockages « anonymes » qui n'ont pas à gérer de sessions. D'autre part, nous avons effectué des tests en remplaçant la liaison nfspd-iods par des connexions TCP mais cette approche s'est révélée non-satisfaisante : TCP fiabilisant la liaison et s'assurant que les iods recevaient les paquets empêchait le traitement des requêtes arrivant sur le serveur, ce qui faisait ré-émettre les clients, générant ainsi plus de requêtes, etc.

En plus des diverses données demandées, les iods doivent pouvoir renvoyer aux clients les informations sur leur méta-données ce qui fait que les informations sur la « pseudo-connexion[5] » client-serveur doivent aussi être transmises.

[5]Nous utilisons le terme pseudo-connexion : ce N'EST PAS une connexion au sens TCP mais cela désigne le couple de valeurs IPsource-IPdestination-PORTsource-PORTdestination qui est utilisé pour gérer les demandes entre le client et le serveur.

Un autre problème que ce protocole doit pouvoir résoudre est le cas où les données demandées par un client seraient « mal-placées », c'est-à-dire situées à cheval entre deux iods. En effet le client dispose d'une vue linéaire du fichier donc rien ne l'empêche d'accéder à des « *offsets* limites » (en pratique ce sont toujours des *offsets* multiples de 4KB pour pouvoir être facilement cachés dans une page mémoire sur x86). L'implantation actuelle utilise des blocs de stockage de 64KB et donc si un client demande à lire 8KB à partir de l'offset 60KB, il va se trouver dans ce cas défavorable. Pour résoudre ce problème, nous avons mis en place une sorte d'entête permettant de gérer un routage par la source.

Puisque nous avons souhaité concentrer la configuration en un seul point (à savoir le méta-serveur), ils nous a semblé logique qu'il remplisse lui même l'entête avec le chemin que la requête doit suivre pour être résolue. Ainsi, pour continuer sur notre requête de 8KB à l'offset 60KB, le méta-serveur va envoyer le message au premier iod en indiquant dans l'entête la référence sur le deuxième iod qui devra être contacté afin de satisfaire la requête.

Le premier iod va recevoir et lire les 4KB qu'il possède. Il va ensuite renvoyer une nouvelle requête similaire à la première comprenant les 4KB déjà lus mais cette fois au second iod qui était indiqué dans l'entête. Le second iod va lire les 4KB restants puis générer la réponse à destination du client.

Certes, ce fonctionnement a le désavantage d'introduire un *hop* de plus et ainsi d'augmenter la latence du système, ce qui a tendance, de par la nature synchrone de NFS, à diminuer les performances vis à vis d'un client. Cependant, comme nous souhaitons le rappeler, il faut conserver en mémoire que l'utilisation envisagée correspond à celle où plusieurs clients accèdent en parallèle aux fichiers et peuvent donc ainsi permettre d'observer un recouvrement des entrées-sorties. Évidemment, le cas pathologique de cette approche serait une application qui lirait uniquement des données aux alentours des adresses multiples de la taille des blocs de stockage.

6.3.1.7 Format des fichiers sur les iods

Les données stockées par les clients sont stockées sur les iods dans un répertoire dépendant du système de fichiers standard (par exemple de l'EXT2). Nous supposerons dans la suite de ce paragraphe que ce répertoire est le répertoire `/data`.

La figure 6.2 page suivante illustre l'évolution que nous avons fait subir aux données sur les iods. La figure de gauche représente un fichier standard tel qu'une application le voit, c'est-à-dire une suite linéaire d'octets.

La figure centrale représente le mode que nous avions retenu pour gérer le stockage dans un premier temps avec l'une des versions du prototype. Chaque bloc de données à stocker correspondait à un fichier sur les iods et ainsi un fichier était stocké sous la forme de nombreux fichiers sur les iods, ce qui peut poser des problèmes car il y avait un très grand nombre de fichiers dans un unique répertoire. Utilisant alors des systèmes n'étant pas particulièrement optimisés pour ce grand nombre de fichiers (EXT2),

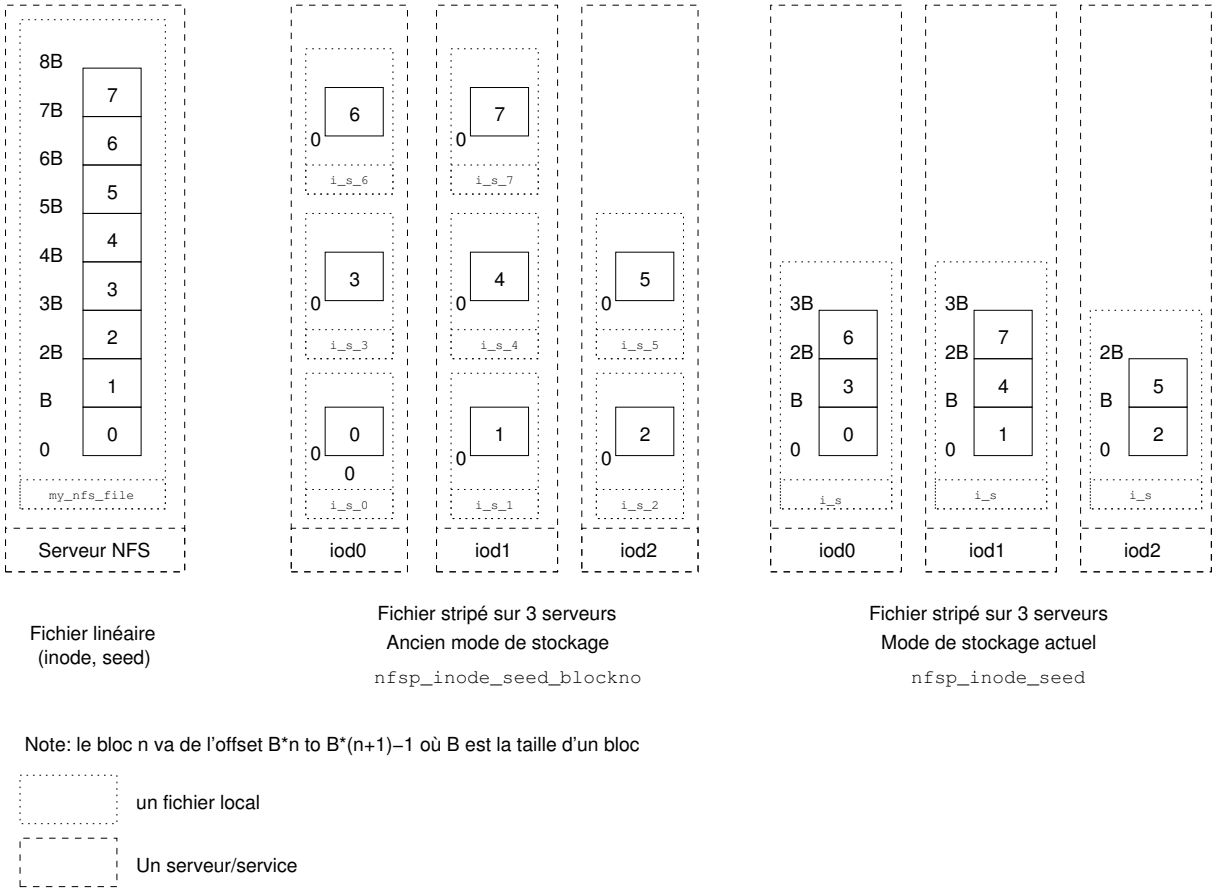

FIG. 6.2 – Format des fichiers sur les iods

nous avons utilisé une technique que l'on retrouve dans plusieurs systèmes de cache en rajoutant des sous-répertoires dans le répertoire de stockage. À titre d'exemple, un fichier d'1GB représente 16384 blocs de 64KB, ce qui se traduisait par la gestion de 2048 fichiers par iod (pour 8 iods de stockage) dans un seul répertoire[6].

Ainsi les données correspondant au méta-fichier `/nfsp/foo` d'inode `0x42` et de graine `0x1234` correspondant à l'offset `0x789000` sont stockées dans un répertoire : `/data/<hash(42,1234,0x789000)>/i00000042_s00001234_o00789000` Ainsi les fichiers sont répartis dans plusieurs répertoires de manière plus équilibrée grâce au bon comportement de la fonction de hachage.

Le problème qui s'est alors posé a été de trouver un moyen efficace pour gérer l'effacement des blocs appartenant à un même fichier, puisqu'avec cette nouvelle répartition, il fallait alors parcourir tous les sous répertoires de `/data` à la recherche des bons noms de fichiers (ceux en `i00000042_s00001234_*`), ce qui bien sûr s'avère être une opération fort coûteuse.

[6]Dans la version d'EXT2 utilisé lors des tests, un répertoire ne pouvait contenir qu'environ 30000 fichiers.

Partant de ce constat, nous avons agrégé les blocs de données sur les iods comme cela est illustré dans la partie à droite sur la figure 6.2. Ainsi, moyennant un léger ajout de logique sur les iods, ces dernieurs pouvaient retrouver les données demandées en fonction de la largeur du *stripe* (le nombre d'iods sur lesquels les données sont réparties de manière cyclique) ainsi que la taille des blocs. Afin de pouvoir gérer ultérieurement des tailles variables et de limiter la quantité de configuration requise sur les iods, nous avons rajouté ces deux valeurs dans les messages du protocole nfspd-iod.

À titre d'exemple, la gestion d'un fichier d'1GB représentant 16384 blocs de 64KB, se traduit par l'utilisation d'un seul fichier de 128MB par iod (toujours dans le cas de l'utilisation de 8 iods de stockage).

C'est ainsi qu'une demande d'effacement de fichier de données venant du serveur de méta-données peut être géré de manière rapide sur l'iod puisqu'il n'y a plus qu'un fichier de stockage sur chaque iod pour chaque fichier réel.

6.3.2 Description du plan d'expériences

Les expériences et mesures effectuées ont été réalisées, sauf mention spéciale, sur l'*i-cluster* (décrit en annexe B page 153) à des époques différentes, avec des configurations des commutateurs Ethernet différentes et avec des variations des versions des noyaux Linux. Toutes les versions font partie de la branche 2.4.x et les versions ont varié des premiers 2.4.9 aux 2.4.21. Il faut de plus noter que ces changements de version des noyaux (plus éventuellement les *patches* des distributeurs) peuvent occasionner des variations de performance importantes dans la mesure où le fonctionnement de la mémoire virtuelle (et donc du cache NFS) est altéré[7].

La plupart du temps, le facteur que nous avons mesuré est la bande passante agrégée : c'est-à-dire la quantité totale de données efficaces transférées pendant la durée de l'expérience. Ainsi, si 8 clients reçoivent un fichier de 1GB en une durée totale de 3minutes, la bande passante agrégée se trouve par l'évaluation de la quantité suivante :

$$BPA = \frac{(8clients) \cdot (1GB/client)}{3min} = \frac{8192MB}{180s} = 45.5MB/s$$

Il est à noter que nous avons à chaque fois considéré le temps de terminaison du dernier client, ce qui a son importance car l'observation, comme nous le montrerons plus loin, a montré que les clients ne finissent pas leurs opérations en même temps même s'ils ont été lancés simultanément (ou du moins dans un laps de temps très faible grâce au lanceur parallèle des Ka-tools[MR01]).

En ce qui concerne le choix de cette métrique qui permet d'avoir une vue des performances globales du système, nous pensons qu'elle est plus pertinente que, par

[7]Par exemple, la branche « stable » 2.4.x des noyaux Linux a connu un changement de mémoire virtuelle dans la version 2.4.10. De plus les différents vendeurs, peuvent aussi ajuster les paramètres de leur noyau afin que ces derniers fournissent une meilleure réponse à l'utilisateur au détriment des performances (*throughput*).

exemple, la bande passante par client, ou bien que le nombre de requêtes traitées par seconde. D'une part, cette mesure peut permettre de calculer les deux autres par des opération simples (division par le nombre de clients ou par la taille moyenne des requêtes réponses), et d'autre part, elle permet de bien voir le gain d'efficacité par rapport à un serveur classique. Dernier point, mais pas le moindre, c'est aussi une mesure souvent utilisée dans d'autres systèmes de fichiers distribués, ce qui permet d'envisager une comparaison avec d'autres systèmes.

Nous donnerons aussi quelques éléments de fonctionnement avec PVFS qui a tendance à fournir de meilleures performances mais au prix, nous tenons à le rappeler, d'une intrusivité plus grande et d'un modèle de fonctionnement très différent (une fois les méta-données récupérées sur un client, il peut contacter directement les iods et effectuer des transferts directs alors que, NFS étant sans état, il faut repasser par le méta-serveur pour chaque requête).

6.3.3 Évaluation de uNFSP

La conception de ce premier modèle n'est bien sûr pas censé pouvoir apporter un gain en écriture. En effet, puisque nous souhaitons complètement remplacer un serveur NFS, les clients ne voient toujours qu'une seule interface d'entrée et donc, dans le cas de l'*i-cluster*, une seule interface à 100Mb/s. Or, puisqu'ils sont censés ne voir qu'un seul et unique serveur pour les raisons de transparence évoquées, ils ne peuvent envoyer les données à écrire que vers celui-ci qui les renverra alors par cette même interface aux serveurs de stockage.

Les performances sont du même ordre de grandeur que sur un serveur NFS équivalent (sur une interface à 100Mb/s bien sûr). La baisse de performances s'explique par plusieurs raisons :

– en NFS2, les écritures étant complètement synchrones, la légère augmentation de latence diminue les performances puisque les coûts de communications ne sont pas récupérés
– le méta-serveur reçoit et ré-émet sur la même interface (le *full-duplex* diminue cet effet mais cela est moins efficace que deux interfaces séparées)

Par rapport à une approche à la PVFS, où le méta-serveur est juste contacté au début des transferts, cette approche par blocs est assez coûteuse car elle requiert pour chaque requête de 8KB de repasser par le méta-serveur, que cela soit en écriture ou en lecture.

Les écritures ne sont donc pas le point fort de cette approche mais à vrai-dire les gains espérés se situent principalement au niveau du fonctionnement de la lecture. Notre hypothèse de départ étant que le temps passé pour traiter la requête sur le méta-serveur est très inférieur au temps passé à satisfaire une entrée/sortie disque complète, et ainsi on peut avoir un certain parallélisme dans ce modèle. Un modèle de fonctionnement similaire peut être vu avec la boucle principale d'un serveur *multi-threadé* qui s'exécute rapidement et utilise un *thread* de travail à chaque nouvelle arrivée de requête ou bien demande de connexion.

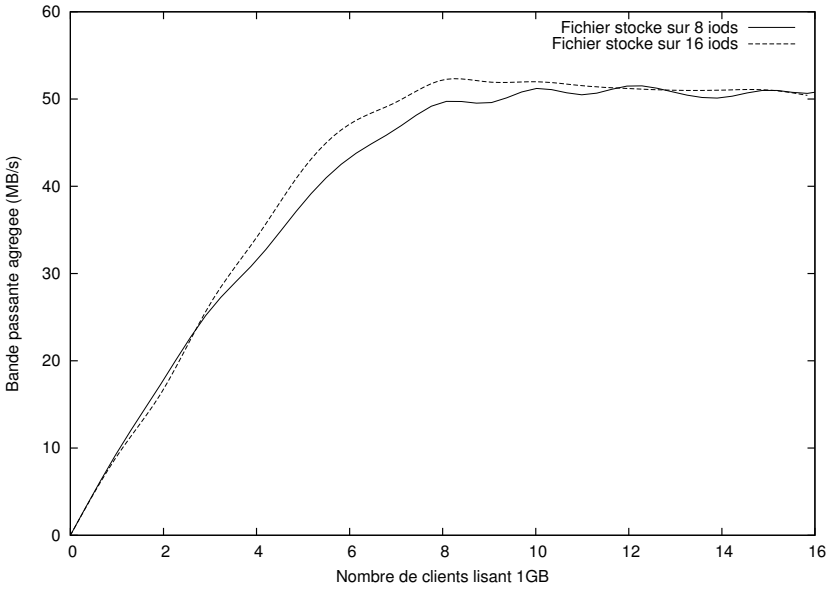

FIG. 6.3 – Performance `READ` avec uNFSP (fichier dans cache)

La figure 6.3 illustre les performances qui sont obtenues avec la lecture parallèle par un nombre croissant de clients lorsque le fichier lu tient dans le cache des iods. Le fichier fait 1GB et est d'abord stocké dans cette courbe sur 8 ou 16 nœuds, puis il est lu de manière concurrente par un nombre croissant de clients, sur la courbe jusqu'à 16 clients parallèles. Les machines avec les iods et les clients sont des ensemble disjoints de nœuds, mais d'autres expériences menées avec iod et client sur la même machine ont affiché des performances à peu près similaires, principalement car le surcoût de gestion occasionné par les iods reste faible.

Cette courbe illustre le fait que l'on observe bien un gain de performances puisque l'on dépasse globalement la barrière des 100Mb/s (soit 12.5MB/s théoriques - la limite pratique étant inférieure). Deux phases sont à remarquer : la première phase correspond à une croissance rapide et la seconde à une stagnation des performances. La première phase correspond à une phase pendant laquelle le méta-serveur dispose de marge pour traiter les paquets mais après (seconde phase) celui-ci commence à avoir son CPU saturé. Nous imputons cette stagnation à plusieurs raisons, mais c'est principalement le fait que le mode de fonctionnement de ce prototype a de nombreux surcoûts dûs principalement aux changements de contextes (occasionnés par de nombreux appels systèmes et interruptions réseau), ce qui est visible lorsque l'on observe le processus avec les outils système car celui-ci passe une grande partie du temps alloué en espace système.

Les iods ne saturant pas (en termes réseaux et charge CPU), augmenter leur nombre n'apporte que peu de bénéfices et se traduit sous la forme d'une faible amélioration des performances : en doublant leur nombre, ceux-ci sont moins sollicités et peuvent donc gérer un plus grand nombre de requêtes.

uNFSP (*user-mode* NFSP)	uNFS (*user-mode* NFS)
fichier d'export avec '`nfsp`' pour les exports nfsp	fichier d'export standard
définir les machines utilisées pour le stockage	-
lancer les iods	-
lancer les services `portmap`, `mount`	idem
lancer le serveur	idem
monter l'export sur les clients	idem

TAB. 6.1 – Comparaison de l'installation d'un serveur uNFSP et uNFS

Dans le cadre de lectures avec des fichiers ne tenant pas dans les caches des iods, les performances sont moins bonnes pour plusieurs raisons. Une des principales est que, justement, puisque les caches ne sont pas utilisables, les disques de ces derniers sont plus sollicités pour récupérer les données, ce qui a pour effet d'augmenter la latence de réponse des iods. Or, ceci diminue les performances globales du système à cause de la synchronicité des accès par NFS. De plus dans de tels cas, le disque peut souffrir de nombreux accès parallèle car la tête de lecture du périphérique doit aller et venir sur le disque pour récupérer les divers fichiers.

Olivier Lobry a réalisé une évaluation plus poussée des performances qui peuvent être obtenues avec NFSP et PVFS. Les résultats de son étude se trouvent à `http://www-id.imag.fr/Laboratoire/Membres/Lobry_Olivier/PVFS_NFSP/PVFS_NFSP.html`.

6.3.4 Exemple d'installation

uNFSP a été conçu pour minimiser les pré-requis d'installation pour les clients (le client standard est utilisé) mais avoir un mode d'installation similaire au programme de base est aussi appréciable. Le mode d'installation détaillé est décrit dans le paquetage, mais ressemble fortement à celui de la mise en place d'un serveur NFS standard et est illustré dans le tableau 6.1.

Les opérations à réaliser pour l'installation d'un serveur NFSP requièrent les mêmes privilèges que celles identiques à réaliser pour l'installation d'un serveur NFS, c'est à dire les droits d'administrateurs (`root`). Ceci est rendu nécessaire notamment car le serveur NFS souhaite que le fichier d'exports appartienne au `root`, et aussi que les services `portmap` et mount requièrent des ports systèmes (< 1024). D'autre part, le serveur, bien que sur un port non-système donc à priori sans avoir besoin du `root`, requiert cependant ce privilège afin de pouvoir gérer les permissions sur les fichiers.

En ce qui concerne NFSP et ses iods, les privilèges du `root` peut être requis (ou pas) selon le fonctionnement des clients. En effet, si le client effectue des vérifications complètes sur les paquets qu'il reçoit, le lancement des iods doit être effectué avec les privilèges de l'administrateur, et ce afin de pouvoir créer et utiliser les sockets de bas niveaux (*socket* IP RAW ou *socket* de type *link layer*) et pouvoir complètement se faire passer pour le méta-serveur.

Cependant dans le cadre de clients Linux comme ceux que nous avons utilisés (noyaux 2.4.[12]x), nous avons remarqué, qu'en fait, ils ne faisaient pas toutes les vérifications auxquelles on était en droit de s'attendre. En effet, tous les paquets qui viennent sur le port UDP correspondant au montage et qui respectent un format NFS avec un `xid` attendu - peu importe leur adresse source et port - sont acceptés ! Ainsi, quelqu'un qui aurait accès « au fil » en *sniffant* le réseau peut très facilement corrompre une session NFS ce qui a tendance à ne pas conforter la mauvaise réputation de ce protocole en ce domaine.

6.3.5 Conclusions sur le premier prototype en mode utilisateur

Les chiffres que nous avons obtenus se sont montrés prometteurs mais le prototype tendait à voir le méta-serveur saturé, ce qui avait tendance à brider la bande passante disponible pour les clients malgré l'augmentation du nombre d'iods. Bien que les performances ne soient pas optimales, le serveur en mode utilisateur conserve malgré tout une certaine utilité, car il peut s'installer sans risque sur une installation existante et ainsi fournir un niveau de performances supérieur à ce que fournirait un serveur simple. Il peut aussi permettre d'avoir un système distribué de stockage permettant d'agréger les espaces disques inutilisés sur plusieurs serveurs sans avoir à toucher aux systèmes installés.

Puisque les coûts d'une application en mode utilisateur devant gérer un grand nombre de messages sont souvent prohibitifs, et ce à cause du grand nombre de changement de contexte requis par les appels systèmes et par les recopies mémoires, nous avons envisagé le passage du méta-serveur en mode noyau afin d'évaluer l'amélioration possible des performances. Ce sont ces travaux que nous présenterons dans la section suivante.

6.4 Implantation en mode noyau

Bien que l'on perde un peu de souplesse en installation (il faut recompiler et charger un module NFS modifié), la recherche des gains de performances espérés nous a mené sur cette voie. En s'appuyant de manière poussée sur le démon NFS kernel Linux déjà existant, un port en mode noyau du prototype utilisateur a été réalisé par Olivier Valentin [Val02] lors de son stage pendant l'été 2002.

Nous avons choisi de rester avec la version 2 du protocole car la gestion de certaines extensions de NFS3 (notamment la gestion du `COMMIT`) requerrait de maintenir un état sur le serveur de méta-données et de mettre en place une barrière de synchronisation, ce qui serait revenu à bloquer le serveur le temps d'effectuer une synchronisation de tous les partenaires de stockage.

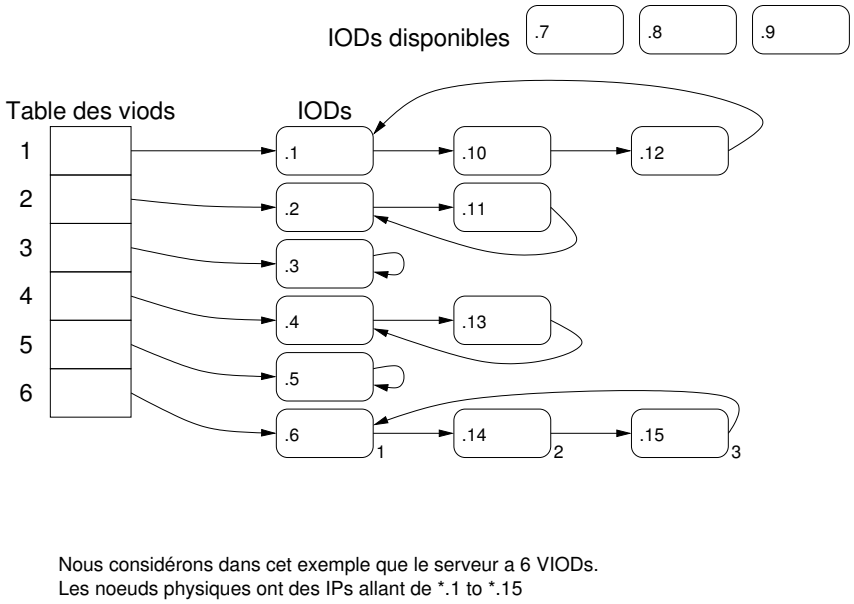

FIG. 6.4 – Principe des viods

6.4.1 Implantation

L'implantation en mode noyau a été réalisée en parallèle avec les tests que nous effectuions avec les extensions de type réplication de données aussi avons-nous souhaité à cette époque avoir un support pour pouvoir gérer le remplacement dynamique d'un nœud de stockage par un nœud qui aurait été répliqué avant une éventuelle panne.

Pour ce faire, nous avons ajouté un niveau supplémentaire d'indirections dans le choix du nœud de stockage sous la forme de viods. Un viod est en fait un groupe d'iods répliqués et qui gèrent leur réplication. Ainsi, lorsqu'une donnée doit être écrite, elle était auparavant écrite sur un iod, mais maintenant elle doit être écrite sur un viod : le message finit bien sûr par être envoyé à un iod qui devient alors le chef du groupe. Le serveur centralisant la configuration du système, il est le seul à connaître tous les membres de tous les viods.

Les viods doivent donc permettre d'implanter une politique simple de tourniquet sur le serveur qui va alors répartir les requêtes sur chacun des membres des viods afin de répartir la charge. Un service de *monitoring* (développé par Adrien Lèbre) permet de reconfigurer le noyau avec une liste d'iods valides ou bien d'en déconnecter certains qui ne peuvent plus répondre et pourraient donc avoir perdu la synchronisation du système et causer des dommages. Divers moyens de gérer cette réplication ont été envisagés et seront évoqués dans la section 7.2 page 117.

Une table de viods est utilisée sur le serveur et conserve alors une référence sur les iods qui sont alors chaînés entre eux comme cela est illustré sur la figure 6.4. Pour pouvoir gérer ce changement dynamique de processus apparemment simple, l'utilisation de plusieurs verrous a été rendue nécessaire pour que ces divers objets puissent être

manipulables de manière sûre par les divers *threads* `knfsd`.

Afin de gérer la configuration, une structure a été ajoutée à l'appel système déjà existant `nfsctl` (qui fonctionne d'une manière similaire à `ioctl` mais uniquement pour la configuration du serveur NFS). Les utilitaires de configuration du système, disponibles dans le paquetage `nfs-utils`, ont aussi été modifiés afin de pouvoir gérer cette extension.

Ainsi, pour pouvoir exporter une partition en mode NFSP, il suffit de rajouter sur la ligne de l'export de la partition le mot clé '`nfsp`' et le serveur noyau modifié gérera correctement l'export. Chacun des fichiers et répertoires de la sous-hiérarchie exportée en NFSP sera alors « *taggé* » par les paramètres de montage, ce qui permettra finalement d'adapter le comportement du serveur en fonction du type d'export (NFSP ou normal).

Les fonctions de gestion des RPC à modifier ont principalement été les mêmes que celles de la version du mode utilisateur, à savoir les fonctions qui manipulent les méta-données de manière à gérer les informations supplémentaires destinées à trouver les données sur les iods. En ce qui concerne la gestion de l'effacement, celui-ci utilise un démon en espace utilisateur et communique avec le kernel par le biais d'un fichier spécial *character device*, qui gère une FIFO des requêtes d'effacement.

L'un des souhaits lors de l'adaptation de NFSP en mode noyau a aussi été de minimiser les modifications à apporter au serveur NFS standard du noyau, afin d'une part de pouvoir profiter d'un code débuggé et maintenu depuis plusieurs années et d'autre part, de pouvoir effectuer la mise à jour pour les nouvelles versions du noyau de manière aisée. D'autre part, nous avons choisi de conserver le même format de méta-données entre la version utilisateur et noyau du serveur, ce qui permet d'utiliser indifféremment les données avec l'un ou l'autre des prototypes.

6.4.2 Évaluation

Le prototype noyau a permis de fortement diminuer les limitations du prototype utilisateur et ainsi d'augmenter les performances du modèle grâce d'une part à la nature *multi-threadée* du serveur noyau et d'autre part aux recopies évitées entre les espace d'adressage noyau et utilisateur.

La figure 6.5 illustre les performances obtenues avec seulement 8 iods et dans ce cas-là l'optimal est très vite approché. Les valeurs optimales correspondent à la bande passante maximale qui pourrait être obtenue si les débits s'agrégeaient sans problèmes (100Mb/s par carte soit 12.5MB/s théoriques, mais dans la pratique nous avons plutôt obtenu 11.5MB/s).

La figure 6.6 illustre les performances obtenues avec 16 iods et dans ce cas-là la faible différence que nous observions entre uNFSP+8 iods et uNFSP+16 iods et cette fois-ci plus importante. La latence réduite du méta-serveur ainsi que son architecture plus performantes sont des raisons de cette augmentation.

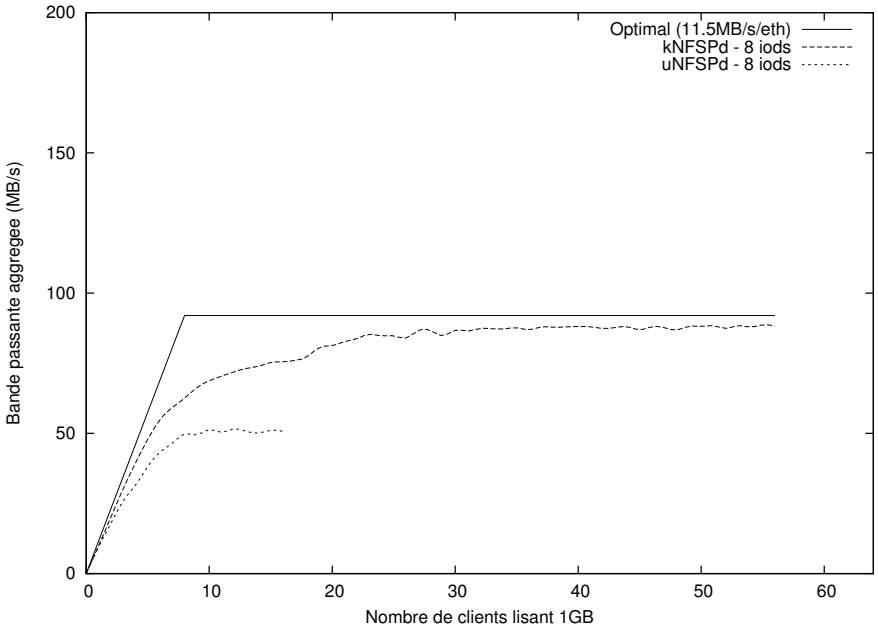

FIG. 6.5 – Comparaison des lecture uNFSP - kNFSP avec 8 iods

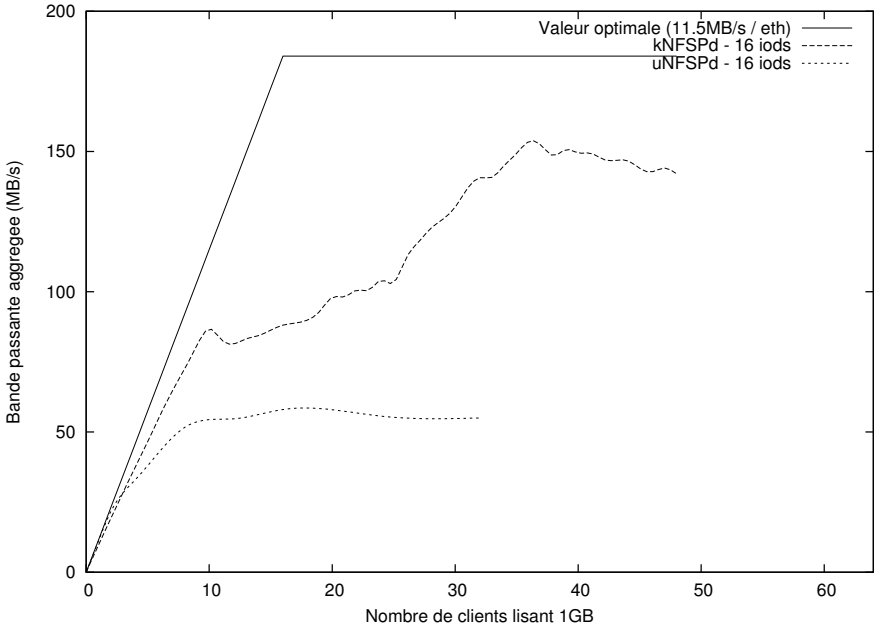

FIG. 6.6 – Comparaison des lecture uNFSP - kNFSP avec 16 iods

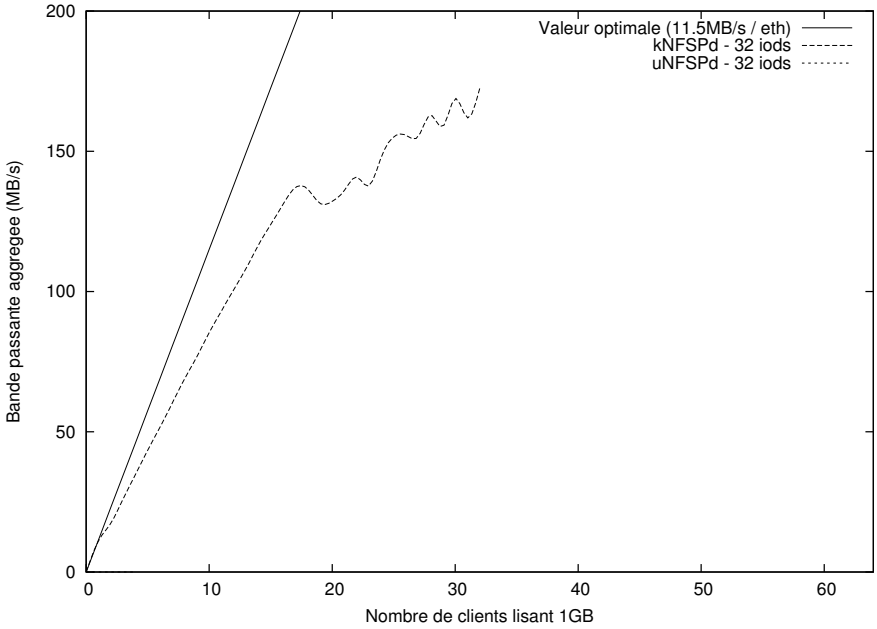

FIG. 6.7 – Comparaison des lecture uNFSP - kNFSP avec 32 iods

La figure 6.7 illustre les performances obtenues avec 32 iods et les performances dans ce cas-là sont meilleures, ce qui s'explique par la moins grande sollicitation des divers iods.

Sur les différentes figures 6.5, 6.6 et 6.7 on peut remarquer que la phase de croissance devient très irrégulière lorsque le nombre de clients devient important. Les figures 6.8 et 6.9, 6.10 fournissent des élément d'explications.

En effet ces figures illustrent les temps de terminaison des différents clients : par exemple pour 10 clients, les 10 croix verticales indiqueront le dates de terminaison de chacun des clients. Pour calculer la bande passante agrégée nous considérons le temps de terminaison du dernier client. Si l'on observe la courbe 6.8, ce temps est assez significatif car les valeurs sont assez regroupées.

En revanche sur les courbes 6.9 et 6.10, les valeurs ont tendance à être beaucoup plus réparties autour de zones, ce qui indique que certains clients finissent très vite et que d'autres ont leur exécution retardée, ce qui correspond à l'apparition de *timeouts* sur le méta-serveur : le nombre de requêtes étant trop grandes, certains clients sont amenés à utiliser leur gestion de la congestion ce qui à tendance à diminuer les performances (mais évite l'effondrement du serveur aussi !)

6.4.3 Exemple d'installation d'un serveur NFSP kernel

La procédure est illustrée dans le tableau 6.2 : une fois le module NFS serveur *patché* et compilé, celui-ci doit être chargé dans le noyau. Il faut aussi installer les outils

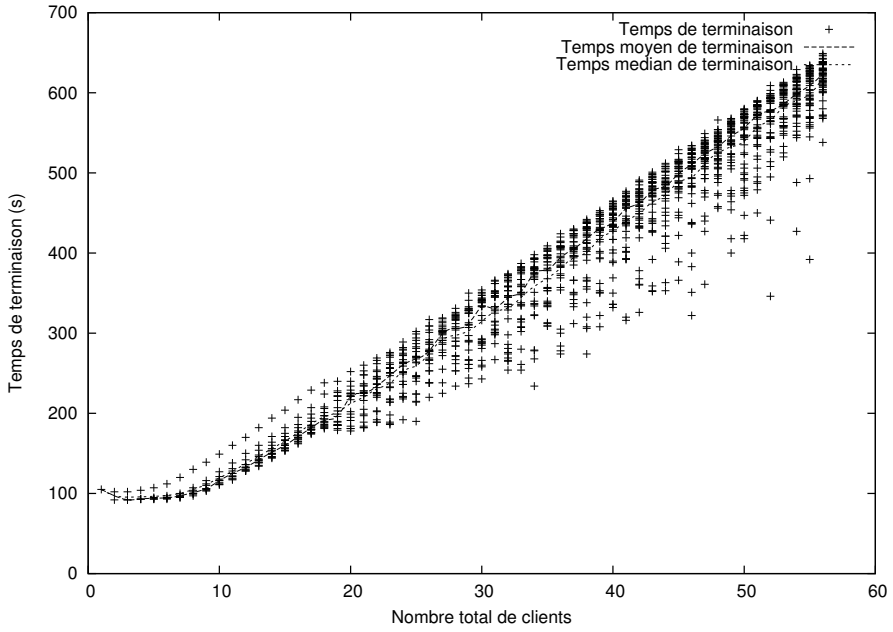

FIG. 6.8 – Temps de complétion des clients avec 8 iods

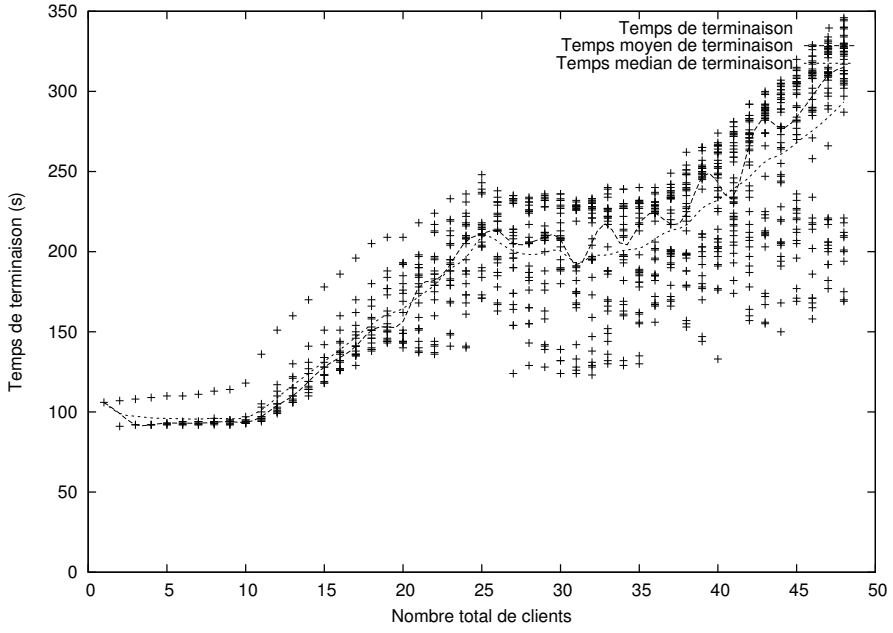

FIG. 6.9 – Temps de complétion des clients avec 16 iods

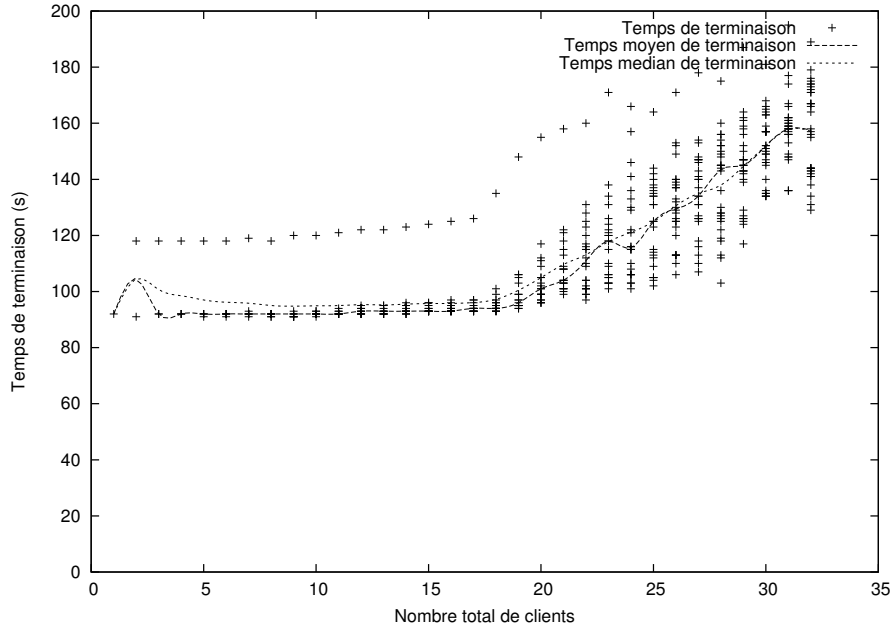

FIG. 6.10 – Temps de complétion des clients avec 32 iods

`nfs-utils` *patchés* qui définissent l'export `nfsp`.

L'utilitaire `exportfs` va alors permettre d'informer le serveur NFS des partitions exportées et de leurs qualités (export standard ou export NFSP).

Afin de configurer le système dans un état de marche, un autre utilitaire est utilisé (`iodctl`) qui établit les bons liens dans le serveur noyau (pour l'instant il définit des viods comprenant un seul iod). Le démon de monitoring `iodmgr` peut aussi être utilisé pour configurer le serveur dans un mode plus avancé (avec des combinaisons différentes d'iods/viods).

kNFSP	kNFS
fichier d'export avec '`nfsp`' pour les exports nfsp	fichier d'export standard
définir les machines utilisées pour le stockage	-
lancer les iods	-
lancer les services `portmap`, `mount`	idem
lancer le serveur	idem
charger la liste d'iod dans le serveur	-
monter l'export sur les clients	idem

TAB. 6.2 – Comparaison de l'installation d'un serveur kNFSP avec un serveur kNFS

6.4.4 Conclusions sur le serveur NFSP kernel

En passant en espace noyau, un gain de performances est donc comme prévu observé mais cela se paie par une complexité d'implantation plus élevée. Les performances sont meilleures à cause des surcoûts de traitement des opérations plus faibles qu'en espace utilisateur mais l'installation doit être réalisée sur une machine spéciale et le système d'exploitation doit être modifié. Certes, cette opération est simplifiée par le fait qu'il s'agisse d'un chargement de module, mais ce dernier peut en cas de *bug* « planter » sévèrement la machine.

De plus, en rentrant dans le noyau, le code devient bien évidemment dépendant de celui-ci et il y a deux choses à maintenir pour faire fonctionner le serveur NFS. Ce sont bien évidemment les *patches* pour le serveur `knfsd` mais aussi les *patches* pour le paquetage d'utilitaires `nfs-utils` (qui lui contient du support pour les noyaux 2.4 et futurs 2.6). Par exemple, des modifications survenues dans une version récente de `nfs-utils` (entraînés par une modification du serveur kNFS dans les noyaux 2.6.x) a entraîné à son tour de nombreuses modifications dans l'implantation de KNFSP (notamment la gestion des communications avec l'espace utilisateur). La conséquence de ceci est que, bien que les modifications d'API interne soient minimes entre deux versions mineures d'un noyau, le portage peut être plus ou moins aisé selon les modifications de ses outils de contrôle en mode utilisateur.

De plus, il est très probable que les bouleversements d'API et de fonctionnement interne qui interviennent à chaque nouvelle version majeure du noyau risquent fort d'entraîner une mise à niveau conséquente du *patch* actuel. Cependant, cet argument est à modérer par le fait que les évolutions majeures restent tout de même peu fréquentes (environ tous les 2 à 3 ans depuis 1996).

Nous avons souhaité conserver les iods en mode utilisateur car le *ratio* entre le gain de performance potentiel et le coût de développement et de maintenance à fournir ne nous a pas semblé suffisamment intéressant. De plus, si l'on souhaite pouvoir les lancer « à la volée », il vaut mieux qu'ils puissent tourner sur des configurations non modifiées.

6.5 Conclusion

Nous venons de présenter dans ce chapitre deux implantations de NFSP réalisées : la première en espace utilisateur, uNFSP, est très peu intrusive sur les clients et les serveurs ; la seconde, kNFSP, requiert des modifications au niveau du système sur le méta-serveur, mais aucune modification n'est requise sur les machines utilisées pour le stockage.

Les performances d'uNFSP sont limitées par plusieurs facteurs qui sont principalement le fait d'être en espace utilisateur et l'architecture non-optimisée du serveur (processus *monothreadé*) ; kNFSP permet d'avoir de meilleures performances au prix d'une

intrusion dans le système d'une machine et bénéficie alors de conditions optimales pour gérer les flux de données (espace noyau et architecture *multithreadée*).

Cependant, comme les différentes courbes ont tendance à le montrer, le méta-serveur reste un point de centralisation, ce qui d'une part ne permet pas de passer sereinement à l'échelle et d'autre part ne permet pas de fournir de système de redondance des méta-données : une panne définitive du méta-serveur entraîne la perte de toutes les données... De même dans le mode actuel de fonctionnement (les données sont *stripées* sur tous les serveurs de stockages), la perte d'un iod risque de rendre toutes les données inutilisables. Cependant, l'introduction des viods permet d'offrir une solution à ce problème en ajoutant de la redondance (comme nous le verrons par la suite).

C'est donc en effectuant ces constations que nous nous sommes orientés sur l'ajout d'extensibilité et de sûreté dans ce système de stockage, aussi allons-nous présenter les diverses expérimentations effectuées dans le chapitre suivant.

Chapitre 7

Extensions et optimisations

Les études menées sur le premier prototype (uNFSP) ont permis d'observer les limites de l'utilisation d'un point de centralisation. Le passage en mode noyau a permis d'optimiser ce fonctionnement mais constitue toujours un point de centralisation et ne fait que retarder l'apparition des limitations. Une solution à ce genre de problème est donc comme toujours de redistribuer la charge. L'une des techniques que nous avons envisagées a été de multiplier le nombre de méta-serveurs. Pour ce faire, nous avons expérimenté plusieurs moyens que nous présenterons par la suite dans la section 7.1.

La multiplication des points d'entrée permet d'autre part d'envisager le terrain de la « sûreté » des données et l'on peut donc commencer à s'intéresser à des voies telles que la haute disponibilité ou plus exactement dans les cas qui nous intéressent à une « bonne » disponibilité, c'est-à-dire en ajoutant un niveau de redondance sur le stockage des données, afin de pouvoir pallier à une panne des machines effectuant le stockage. Nous avons dans un premier temps effectué la mise en place de ces opérations de redondance des données avec le prototype uNFSP et nous présenterons les diverses techniques que nous avons évaluées.

7.1 Multiplication des points d'entrée

Multiplier les points d'entrées afin de pouvoir gérer plusieurs serveurs NFS qui coopéreraient posent de nombreux problèmes si l'on souhaite que le client puisse conserver une vue unique, moyennant la cohérence relâchée du système.

L'architecture alors envisagée se rapproche donc d'une architecture ressemblant « de loin » à celle d'une architecture à disques distribués avec une séparation du système en couche de stockage (les disques $\Leftrightarrow$ les iods) et couche de gestion des méta-données (les serveurs$\Leftrightarrow$ les méta-serveurs), comme cela est illustré sur la figure 7.1 page suivante.

Afin de valider une telle approche nous avons commencé par étudier les divers problèmes qui pouvaient se poser dans le cas de la mise en place d'une telle architecture.

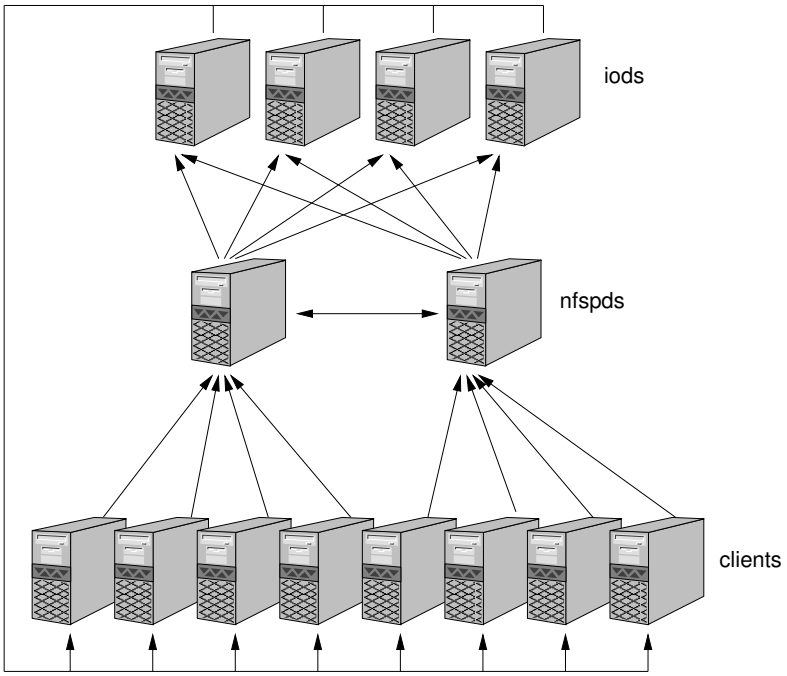

FIG. 7.1 – Multiples serveurs nfspd

Nous décrirons quelques essais préliminaires menés avant de nous concentrer sur la mise en place d'une telle architecture.

7.1.1 La multiplication des points d'entrée pose plusieurs problèmes

Nous situant dans le cadre d'une cohérence temporelle, c'est-à-dire avec relativement peu de contraintes, notre approche semble offrir de nombreux avantages :

- avoir plusieurs points d'entrée permet de diminuer la charge sur les serveurs et donc d'en augmenter les performances. Les solutions proposées par l'industrie proposent souvent la mise en place d'une boîte noire qui équilibre la charge sur différentes machines derrière elle.
- avoir plusieurs serveurs peut permettre d'offrir un certain niveau de résistance aux pannes. Pour avoir un niveau satisfaisant, et ne plus avoir d'unique point de défaillance, il faut que les méta-données et les données soit répliquées : cette technique est par exemple utilisée dans CEFT-PVFS pour le système PVFS. La nature statique du client NFS (un client est associé à un serveur) requiert cependant qu'un serveur qui fonctionne se fasse passer pour le serveur en panne.

Cependant, ces avantages ne vont pas sans certains inconvénients :

- Dans l'implantation actuelle, les fichiers sur les iods sont repérés par le numéro de l'inode du méta-fichier qui les décrit ainsi que par un nombre généré aléatoirement.

Cependant cette approche ne permet pas d'avoir plusieurs serveurs qui partageraient un même inode car il y a fort peu de chances que deux fichiers sur deux machines différentes se voient attribuer le même numéro d'inode.

- Il n'y a plus un seul serveur NFS, mais les clients doivent être répartis parmi les divers serveurs existants. Dans certains cas pathologiques, c'est-à-dire si les clients qui effectuent les entrées sorties sont tous localisés sur un seul serveur, l'approche ne va pas permettre d'apporter de gain de performances, et risque même probablement d'entraîner une légère perte de performances dans la mesure où la cohérence avec les autres serveurs de méta-données doit être gérée.
- La création d'un fichier est une étape « critique » dans la mesure où il faut éviter que deux fichiers différents mais de même nom et dans la même place dans l'arborescence puissent être créés simultanément sur deux serveurs de méta-fichiers. Des algorithmes distribués d'élection peuvent être mis en place mais il s'avère que le fonctionnement du modèle NFS requiert de conserver une latence faible pour ne pas avoir de ré-émissions des requêtes, aussi l'utilisation de tels algorithmes doit plutôt être évitée car elle risque de mener à des créations de fichiers très lentes, ce qui risque de bloquer plus ou moins le serveur.
- L'aspect transparence est aussi un aspect important et donc nous avons choisi de conserver le support des liens durs, ce qui laisse croire à l'utilisateur qu'il utilise un serveur NFS complètement normal.

Dans un premier temps, diverses solutions utilisant des serveurs uNFS natifs ont été déployées et testées. Nous allons maintenant présenter les résultats obtenus avec ces approches.

7.1.2 Approche : mélange d'exports et ré-exports

Le méta-serveur constitue un point de centralisation du système, aussi a-t-il été envisagé de s'affranchir de son unicité pour améliorer les performances. Le modèle retenu pour pouvoir tester cette approche tout en maintenant une cohérence de type NFS a été mis en place lors du déroulement du DEA d'Adrien Lebre. Le lecteur pourra se reporter à [Leb02] ou [LDVL03] pour les explications plus poussées.

Le principe consiste à mettre en place plusieurs serveurs de méta-données (uNFSPd) en parallèle avec des serveurs de données (nfsd). Les méta-serveurs interprètent alors les méta-fichiers qu'ils ont, que ces derniers soient stock, és dans le système de fichier local ou bien qu'ils soient disponibles à travers un montage NFS de la partie des méta-fichiers d'un autre méta-serveur. Ce fonctionnement est illustré pour deux méta-serveurs dans la figure 7.2 afin de ne pas surcharger le schéma mais plusieurs méta-serveurs peuvent être utilisés. Sur cette figure les 4 clients sont répartis sur les 2 méta-serveurs qui partagent les 4 iods pour stocker les données.

Les tests ont pu être réalisés avec les serveurs NFS et le prototype NFSP en mode utilisateur. En effet, l'utilisation d'autres serveurs n'aurait peut-être pas été possible car elle nécessite de pouvoir ré-exporter en NFS des agrégats de partitions de plusieurs types,

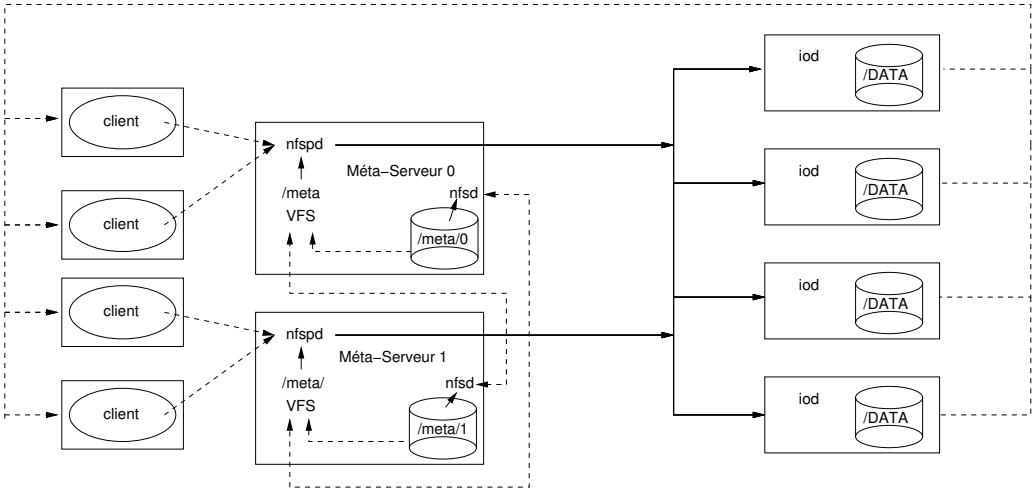

FIG. 7.2 – Utilisation d'exports et de ré-exports

ce que tous les serveurs NFS ne supportent pas (par conception ou par spécification pour les serveurs NFS3). C'est par le biais d'un jeu de scripts *shells* en alliance avec l'utilisation des *ka-tools*[MR01] qu'une telle configuration a été mise en place.

Les avantages observés ont été ceux espérés à savoir de meilleures performances puisque les méta-données sont réparties sur plusieurs machines (voir figure 7.3).

Moyennant le rôle joué par les caches NFS de données et méta-données une unicité d'accès peut être atteinte mais on ne dispose pas vraiment d'une unicité de nommage. De plus cette approche ne permet pas de résoudre plusieurs problèmes évoqués dans la section 7.1.1 page 112.

Un autre problème est que la transparence de la localisation dans le nommage a été perdue puisque les fichiers sont stockés dans un chemin du style `/nfsp/meta-serveurN/fichier`. Ce mode de nommage ressemble alors fortement au système utilisé dans MFS (*Mosix File System*) et est décrit dans la section 4.1.3 page 47.

De plus un autre inconvénient de l'approche retenue est qu'elle ne s'avère pas portable principalement pour cause de non-support des ré-exports de partitions composées de différentes partitions (ou de montages). Par exemple, le serveur en mode noyau n'est pas en mesure d'effectuer cela ; de plus, les normes NFS 3 et 4 interdisent le support de cette fonctionnalité.

En revanche, en cas de panne de l'un des méta-serveurs, on ne perd plus toute la zone de stockage des méta-données mais uniquement les méta-données qui dépendaient du serveur défaillant donc l'accès au reste du système peut continuer à être fourni.

Comme nous venons de le voir, cette approche apporte des gains de performance prometteurs mais a quand même quelques inconvénients aussi avons nous dû nous orienter sur d'autres approches pour essayer de limiter les inconvénients liés à la multiplication des méta-serveurs.

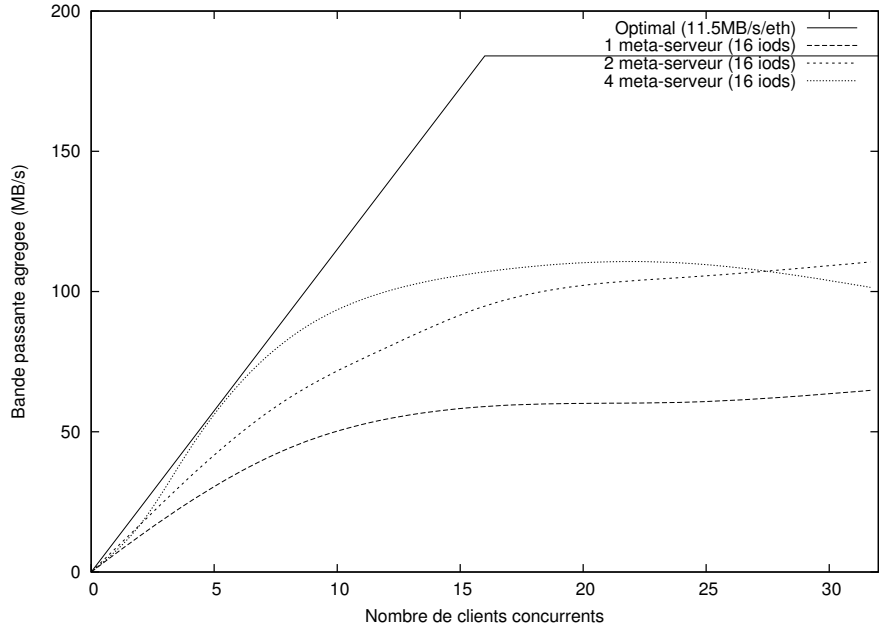

FIG. 7.3 – Évaluation d'une approche mêlant exports et ré-exports

7.1.3 Approche par hiérarchisation du stockage des méta-données

Dans cette approche, le protocole de partage des méta-données est constitué par un autre serveur NFS (*meta-master*) qui exporte une unique partition (voir l'exemple pour deux méta-serveurs dans la figure 7.4).

Chacun des méta-serveurs « normaux » monte alors par le biais de NFS l'export du méta-serveur maître et ré-exporte ces données en NFSP aux clients. Le gain de performances observable se trouve dans l'utilisation des caches NFS des méta-serveurs (qui sont alors des clients du méta-serveur maître). Le méta-serveur maître peut être un serveur NFS classique en mode utilisateur ou bien en mode noyau.

Les méta-données étant de petite taille et cachées par la partie NFS client, la charge occasionnée sur le méta-serveur maître ne devrait pas entraîner une charge trop importante sur ce dernier.

Une évaluation de cette approche doit être effectuée afin de la valider, mais nous pensons que les résultats obtenus devraient être comparables aux résultats de l'approche précédente, tout en résolvant un certains nombre de problèmes, notamment la disparition de la transparence de la localisation du nommage, et le fait que seul le méta-serveur maître doit être rendu redondant si l'on souhaite conserver les méta-données en cas de pannes (et non chaque méta-serveur).

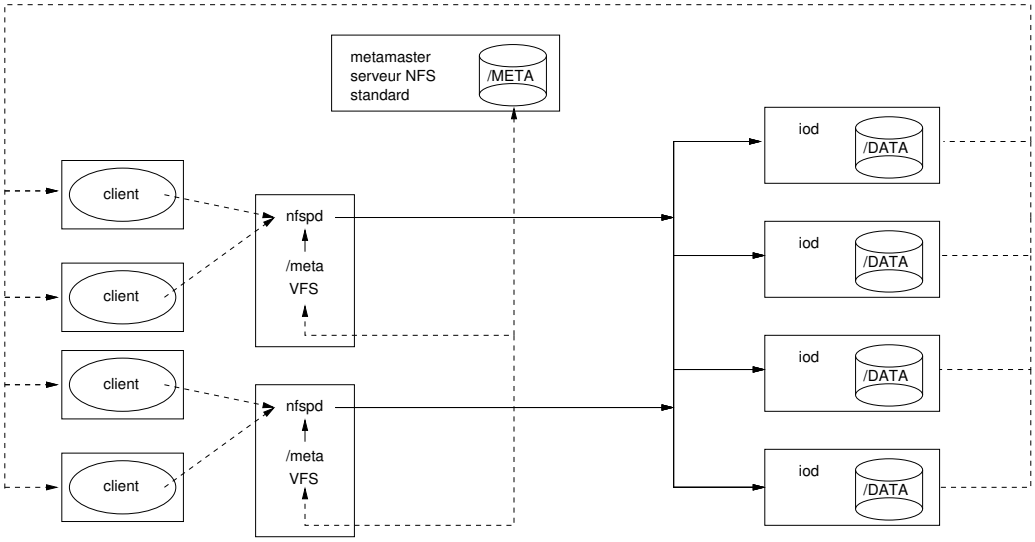

FIG. 7.4 – Utilisation d'un méta-serveur maître

7.1.4 Conclusions sur les différentes techniques de multiplication des points d'entrées

Nous venons donc de présenter deux méthodes permettant de mettre en œuvre une multiplication des points d'entrées : la première consistant sur un mélange d'exports et ré-exports et la seconde sur une hiérarchisation du stockage des méta-données en recourant à un serveur NFS annexe.

Ces solutions permettent d'améliorer les performances si les clients sont bien répartis parmi les différents méta-serveurs. En revanche, la nature statique des clients (un client est associé à un serveur particulier) fait qu'il n'est pas aisé d'offrir de la haute-disponibilité à moins de recourir à des techniques de reprise de service (par forçage des caches ARP). Dans un tel cas il est nécessaire de gérer un mécanisme de reprise après faute pour réintégrer les serveurs fautifs.

D'autres techniques que celles que nous venons de présenter peuvent bien sûr être envisagées en modifiant l'implantation de NFSP « en dur » pour par exemple mettre en place une cohérence directement au niveau de la gestion des méta-données : c'est une piste de recherche que suit actuellement Rafael Ávila dans le cadre de sa thèse au laboratoire ID-IMAG.

D'autre part, en ayant plusieurs points d'entrée, seule une partie du chemin est effectuée car il manque la redondance au niveau des données, aussi allons-nous présenter les approches que nous avons envisagées pour gérer cet ajout de redondance.

7.2 Support de la réplication des données

La réplication permet par la redondance des informations qu'elle met en place de contribuer à fournir un service plus sûr et de tolérer les pannes de machines tout en préservant les données. Cette section présente nos diverses expérimentations dans ce domaine.

7.2.1 Réplication et NFS

De par son omniprésence, et son protocole « fortement connecté », dans le sens où si le serveur tombe en panne, le client ne peut plus fonctionner, plusieurs solutions ont été développées par le passé pour permettre aux divers clients de continuer à bénéficier du service de fichiers malgré les défaillances du serveur. Dans la plupart des cas, les solutions développées offraient une réplication forte par le biais de matériel spécialisé (dans HA-NFS [BEM91] les disques sont partagés entre deux serveurs qui se relaient en cas de panne) ou bien des protocoles de cohérence forte entre les différents serveurs mirroirs (par le biais de réplication active dans [LGG+91] ou par le biais de transaction et de caches persistants dans FT-NFS [PM96]).

Cependant ces approches visent principalement à faire de la haute-disponibilité : c'est-à-dire que le serveur ne doit jamais être arrêté même lorsqu'une panne est détectée. Cependant, bien que le taux de pannes du matériel ait tendance à diminuer, pour garantir un service de haute disponibilité, la nécessité de matériel spécifique se fait souvent sentir afin de détecter les pannes de manière suffisamment fiable (malgré l'impossibilité théorique d'un consensus distribué [FLP85]). Ainsi, en utilisant plusieurs réseaux en parallèle (un réseau usuel et un réseau d'administration et/ou de surveillance), on peut avoir l'« intuition » que si les deux réseaux ne répondent plus alors la panne doit être plus probable . Une fois la panne détectée, il faut alors se faire passer pour le serveur en panne (seules les pannes franches sont considérées), ce qui se fait de manière usuelle par forçage des caches ARP des machines dans le cas de serveurs sur réseaux de type Ethernet.

Les projets comme HA-LINUX[Rob] (*High Availability Linux*) fournissent diverses briques de base pour mettre en place des serveurs redondants hautement disponibles (surveillance, détection de pannes et désactivation définitive[1] des nœuds en pannes, ...)

Cependant dans la plupart des cas, les solutions sont relativement intrusives et concernent un petit nombre de répliques qui doivent être fortement cohérentes. Dans le cas de NFS, il faut conserver en mémoire que tant que le client n'a pas reçu d'acquittement, il va réessayer de faire passer son message. De plus NFS, n'est pas vraiment optimisé pour la gestion des conflits écriture/écriture entre deux clients ou bien écriture/lecture

[1] Nous espérons qu'il s'agit d'un euphémisme car le module s'appelle STONITH (*Shoot The Other Node In The Head*) et à notre connaissance, les fabricants ne semble pas fournir de matériel pouvant gérer cette extension dans sa version littérale (quoique de nombreux périphériques puissent être disponibles de nos jours sur ports USB... 8-)

par deux clients différents, à cause de la gestion indépendante des caches sur la partie cliente.

Des études menées lors du développement de CODA ont montré que très peu de conflits écriture/écriture se sont produits lors des traces d'utilisation ayant été considérées (pour mémoire, dans ce système supportant un mode déconnecté, la réconciliation peut avoir à être effectuée manuellement par l'administrateur). D'autre part, nous pensons que les utilisateurs qui utilisent et programment des applications évitent de tels comportements propices à une non-portabilité de leur application. En effet, à moins de prendre des verrous, la sémantique POSIX ne dit pas qui doit avoir raison, lorsque deux processus font des écritures parallèles sur une même zone d'un fichier (en revanche si un processus effectue une lecture après une écriture, il doit voir la dernière valeur - ce qui explique pourquoi NFS n'a pas une sémantique POSIX [caches clients indépendants]).

Le mode de fonctionnement que nous envisageons est donc celui d'une application lisant des données et en produisant, mais dans un autre fichier qui peut alors être utilisé par une autre application, etc... À moins de disposer d'un système spécifique pour une grappe Beowulf toute cliente d'un gros SAN avec un DFS pour disques distribués (GFS/GPFS/...), nous ne pensons pas que ce problème bénéficie de solutions satisfaisantes.

Pour revenir au domaine de la réplication, nous nous sommes donc efforcés de mettre en place un système de redondance permettant d'offrir la défaillance de serveurs de données (iods) de manière transparente pour les clients (ils n'ont de toute façon pas conscience de l'existence de ces derniers). Nous nous autorisons un arrêt du système (plutôt un gel des clients) pour une remise en cohérence des réplicats après une panne. Le but est de continuer à offrir un service au moins équivalent à NFS en terme de cohérence des données, de même que de ne pas renvoyer aux clients des données périmées, tout en ayant un faible surcoût de réplication.

Un autre bénéfice que nous pourrons tirer de la réplication des données sera aussi de pouvoir avoir de meilleurs performances si l'on peut fournir un service du type écrire et lire les données sur n'importe lequel des iods car cela diminuera leur charge (notamment pour des lectures).

7.2.2 Réplication « simple »

La réplication peut être obtenue en envisageant de répliquer les données stockées sur les iods. Plusieurs techniques sont envisageables, aussi allons-nous les présenter, puis nous présenterons celle que nous avons finalement retenue, compte tenu des spécificités des comportements des clients NFS (ré-émissions jusqu'à acquittement).

Nous considérerons à chaque fois que le nombre de réplicats est relativement petit (2 ou 3 nous semble un nombre réaliste) pour plusieurs raisons. La première est que si l'on considère que les pannes des iods au sein d'un viod sont indépendantes (et faibles), la probabilité que tous les iods soient en panne et qu'en plus l'administrateur

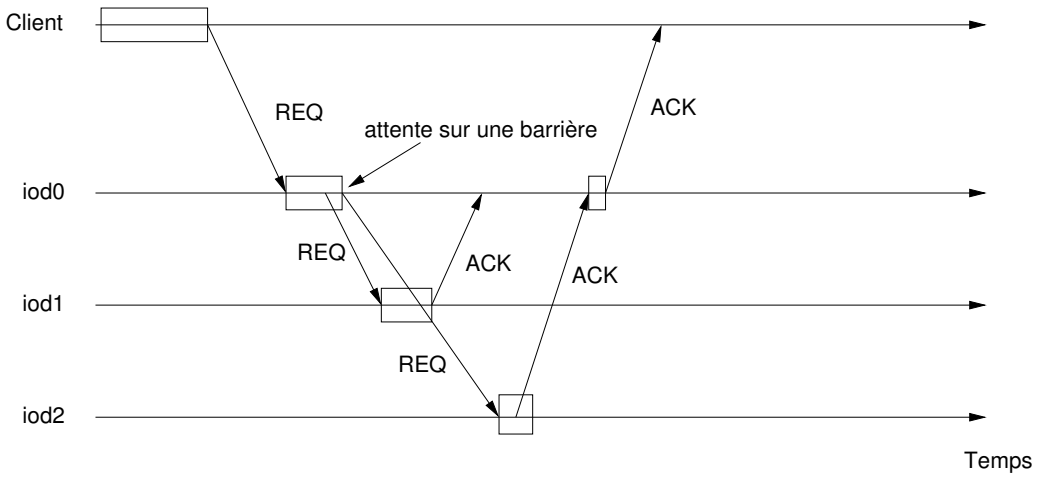

FIG. 7.5 – Technique de réplication avec serveur primaire

ou l'application de surveillance n'ait pu réagir est encore plus faible[2]. D'autre part, à chaque nouvelle réplication, l'espace utilisé est en quelque sorte perdu pour le stockage au profit de la redondance. Partant de tous ces faits, l'utilisation de 2 ou 3 réplicats nous a semblé être un compromis réaliste comme nombre de copies à considérer.

7.2.2.1 Principe

Si nous considérons que le méta-serveur est sécurisé, un moyen d'obtenir de la redondance dans ce système est d'ajouter un système de réplication au niveau des iods.

Deux techniques de gestion de la réplication par le logiciel sont décrites dans [GS97] et illustrées sur les figures 7.5 et 7.6.

Le mode qui se prêterait le mieux au fonctionnement de NFSP correspond au mode avec serveur primaire (figure 7.5) car il permet de transférer la gestion de la charge sur un des iods au lieu de la laisser sur le client (ici le serveur nfspd) comme c'est le cas dans la réplication active (figure 7.6 page suivante).

Cependant une telle technique peut poser problèmes dans le cas de NFSP car la latence doit être conservée la plus faible possible si l'on ne veut pas dégrader les performances de manière trop importante. Ce schéma peut être simplifié si l'on prend en compte la nature des clients NFS : ceux-ci ré-émettent la requête tant qu'elle ne peut être satisfaite. Le modèle simplifié est donc illustré sur la figure 7.7 page suivante.

L'ordonnancement des requêtes illustré sur la figure ' 7.7 page suivante permet ainsi de ne servir que les données les plus « fraîches » aux clients qui les lisent.

[2]Elle est toutefois existante comme par exemple en cas de panne du système de climatisation d'une salle machine en plein été… :-)

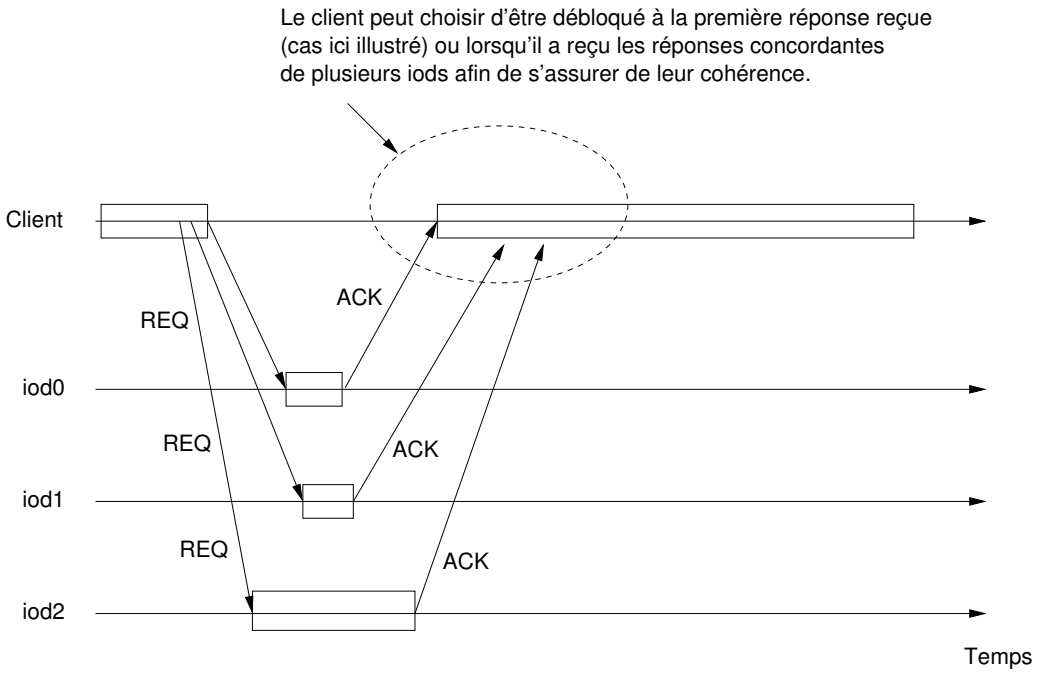

FIG. 7.6 – Technique de réplication active

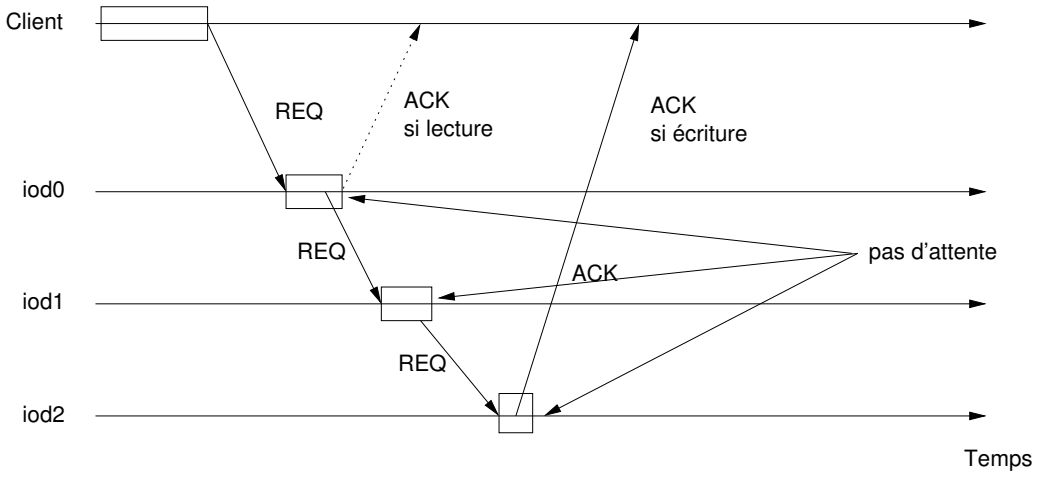

FIG. 7.7 – Réplication « approchée » mise en place dans NFSP

7.2.2.2 Implantation

L'implantation de cette fonctionnalité a nécessité de modifier quelque peu le protocole NFSPD-IOD. Un message à été rajouté dans l'entête des données et le code des iods a été adapté pour gérer l'envoi des données à l'iod de stockage suivant.

Le méta-serveur remplit donc un champ contenant la liste des iods du viod puisque c'est lui qui détient la connaissance exhaustive des membres actifs du groupe. Cette connaissance exhaustive peut être mise à jour par le biais du système de monitoring des iods.

Si un iod ne répond pas à un certain nombre de messages celui-ci est écarté définitivement du groupe de réplication et une intervention manuelle de l'administrateur est alors requise afin de réparer et resynchroniser la machine avec son groupe viod.

7.2.3 Réplication logicielle « intelligente »

Le mode de réplication simple que nous venons de décrire est relativement peu performant car si l'on ne souhaite pas avoir des problèmes de versions, on ne peut pas lire au hasard (ou sur une base de tourniquet) sur plusieurs disques, car on pourrait avoir selon l'ordre des requêtes des versions de données différentes entre deux lectures, c'est-à-dire servir des données anciennes. Pour ce faire, nous avons envisagé un système un peu plus complet pour gérer du « versionnage » des blocs de données.

7.2.3.1 Description

Le principe de cette réplication est que, tout en conservant un mode de réplication relativement simple, on puisse mettre en place un système dans lequel les viods n'auraient plus de vrai *leader*. Le méta-serveur enverrait alors les requêtes de lectures et d'écritures sur une politique de *round-robin* aux membres du viod, ceux-ci se chargeant alors de gérer la duplication des informations avec les autres iods.

Pour ce faire, à chaque bloc constitutif du fichier un numéro de version est associé. Ce numéro peut, par exemple, être calculé en divisant *offset* par la taille d'un bloc de transfert. Les requêtes de lecture et d'écritures vont alors devoir récupérer ce numéro de version avant d'être transmises aux membres du viod. Ce seront ces derniers qui s'assureront de servir des versions « fraîches » des données. Le principe de fonctionnement de cet algorithme est repris sur la figure .

Un des problèmes qui va aussi se poser va être de gérer l'association numéro de bloc - version du bloc sur le méta-serveur et sur les iods. Après avoir étudié différentes solutions pour la gestion de telles structures, nous avons pensé qu'une simple structure linéaire serait la structure adéquate pour gérer ce modèle pour plusieurs raisons :

– simplicité de mise en œuvre
– meilleur compromis entre vitesse d'accès et mémoire à utiliser

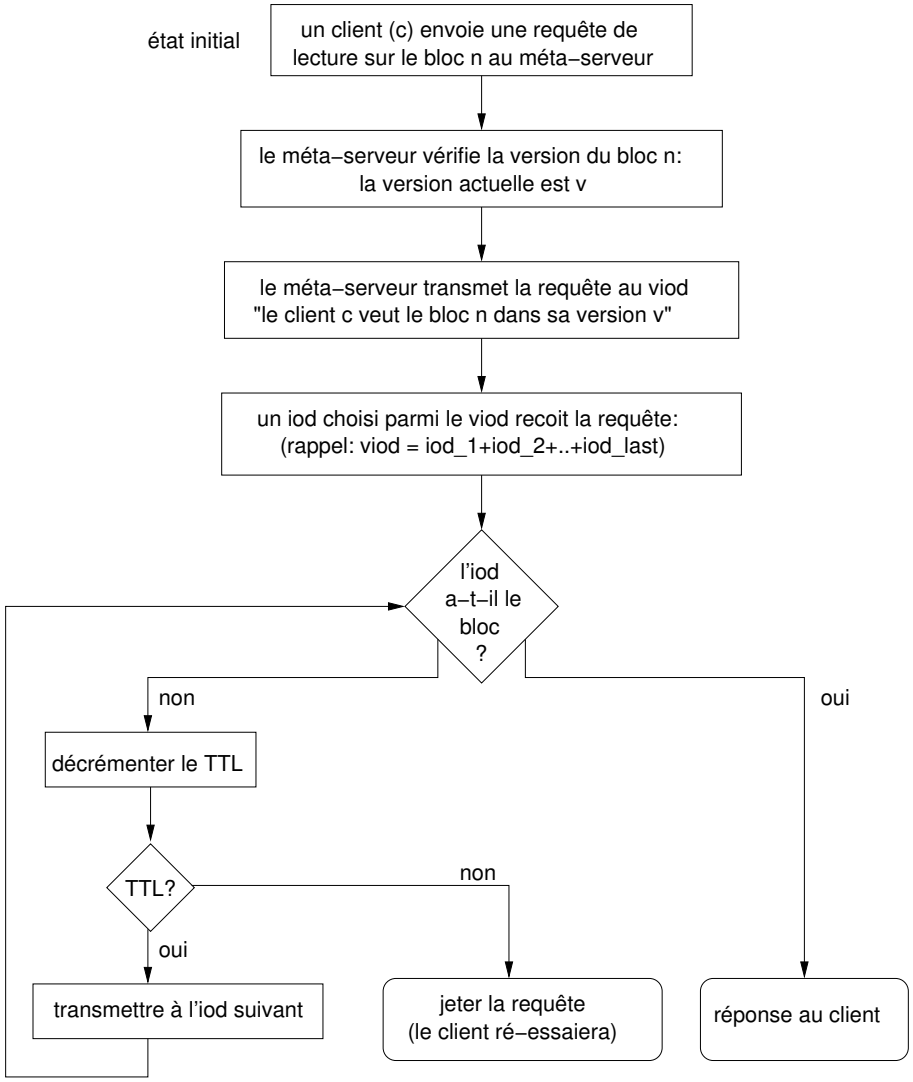

FIG. 7.8 – Fonctionnement de la réplication « intelligente »

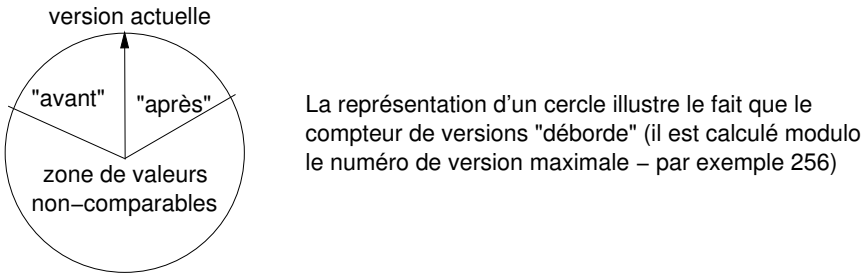

FIG. 7.9 – Intervalle de confiance des numéros de versions

Ainsi, en utilisant par exemple 1 octet pour indiquer la version d'un bloc de 4KB, le surcoût en espace de stockage est relativement minime (moins de 0.03%).

Nous avons aussi pensé utiliser des structures plus complexes à base de *log* ou bien d'arbres, mais l'un des problèmes qui apparaissait était qu'à la création, cette opération allait pousser ces systèmes à leurs limites en « dépliant » les arbres et qu'après, un recompactage serait nécessaire. De plus en cas d'accès aléatoire à un bloc, il aurait pu être nécessaire de complètement déplier la structure de stockage (pour les structures à base d'arbres), ce qui à notre avis aurait été trop coûteux en terme de temps à gérer. Certes l'espace de stockage requis peut être plus petit mais comparé aux surcoût des mauvais cas, cela nous a semblé un compromis acceptable.

Afin de pouvoir gérer le *rollover* des numéros de version, il a été décidé de recourir à un intervalle de confiance comme illustré sur la figure 7.9.

Ainsi, si l'iod n'a pas la version demandée v, il peut effectuer quelques opérations visant à juger la qualité du lien et/ou son propre état. En effet, en évaluant la différence entre le numéro de version demandé et la version dont il dispose pour un bloc, il peut savoir si son état est relativement à jour ou bien pas du tout.

Il faut toutefois noter que pour que cette différence ait un sens, les valeurs ne doivent pas trop différer (*modulo* le nombre maximal de numéros de versions) pour pouvoir être comparables. Si ce n'est pas le cas alors il peut s'être produit un problème de communications, et certaines mises à jour ont pu être perdues. Dans ce cas là, il vaut peut-être mieux que l'iod se saborde en ne répondant plus aux requêtes, afin d'être retiré du viod, en attendant une réaction de l'administrateur qui peut forcer une re-synchronisation du système. Si l'iod reçoit plusieurs fois de suite des demandes de versions non-comparables avec la version qu'il a, la cause la plus probable est qu'il soit complètement désynchronisé avec le méta-serveur.

Une évaluation des gains qui peuvent être obtenus doit aussi être effectuée afin de valider (ou non) cette approche quelque peu plus complexe. Des problèmes peuvent se poser si deux clients effectuent des écritures simultanément sur des morceaux de blocs mais cela se serait produit d'une façon similaire car les conflits écriture-écriture n'ont pas usuellement une sémantique bien définie, et ce, même sur un système de fichiers local.

7.3 Conclusion

Nous venons de présenter dans ce chapitre deux étapes pour la mise en place d'un système de stockage distribué plus sûr :

– la réplication des serveurs de méta-données
– la réplication des serveurs de données : un mode simple et un mode un peu plus optimisé

Le but n'étant pas d'avoir un système à haute disponibilité (le système peut éventuellement être interrompu pour reconstruction ou autres opérations administratives) mais un système qui supporte les défaillances de machines (pannes par exemple) de la ma-nière la plus transparente possible pour les utilisateurs.

Les système de reprise transparente au niveau des méta-serveurs n'ont pas été particulièrement étudiés car le but principal était d'augmenter la performance globale du système mais si le besoin s'en fait sentir, le recours à des systèmes disposant de caractéristiques de haute disponibilité peut être envisagé.

Chapitre 8

Validation

Ce chapitre décrit les applications qui ont été amenées à utiliser nos prototypes ainsi que plusieurs développements actuellement en cours. Nous présenterons d'abord l'application qui a motivé de nombreux choix et contraintes architecturales (distribution des données par exemple), c'est-à-dire l'application de transfert parallèle. Par la suite nous présenterons, une autre application décrivant un système de gestion de fichier peu intrusif pour grappes de grappes.

8.1 Utilisation sur WAN

Depuis son début, NFSP devait constituer une brique de base pour pouvoir réaliser des transferts efficaces de grappes à grappes. Après nos tests en local, une mécanique d'un niveau plus élevé devait être mise en place afin de gérer ces transferts. Ce sont ces mécanismes que nous allons décrire ici.

8.1.1 Rappel du contexte

L'idée sous-jacente au développement de notre prototype était de pallier au problème des transferts efficaces entre deux grappes d'ordinateurs « standards » reliées par une liaison haut-débit, comme ce que l'on peut voir souvent voir dans les grilles de grappes (voir figure 8.1 page suivante).

Si les grappes se font mutuellement confiance, comme c'est le cas dans notre protocole nous disposons alors d'une visibilité IP de bout en bout et donc au lieu de passer par un point centralisateur, il est tout de suite plus intéressant de mettre en œuvre des transferts directs de serveurs de stockage à serveurs de stockages comme l'on peut l'observer dans les systèmes hauts de gamme à la HPSS[HGFW02] ou bien à la DPSS[Lab] « du pauvre » (sans gros serveurs).

L'idée dominante de ces extensions est de fournir une API qui permette d'accéder aux méta-données et de mettre en relation les entités de stockage distantes. Pour ce faire, la

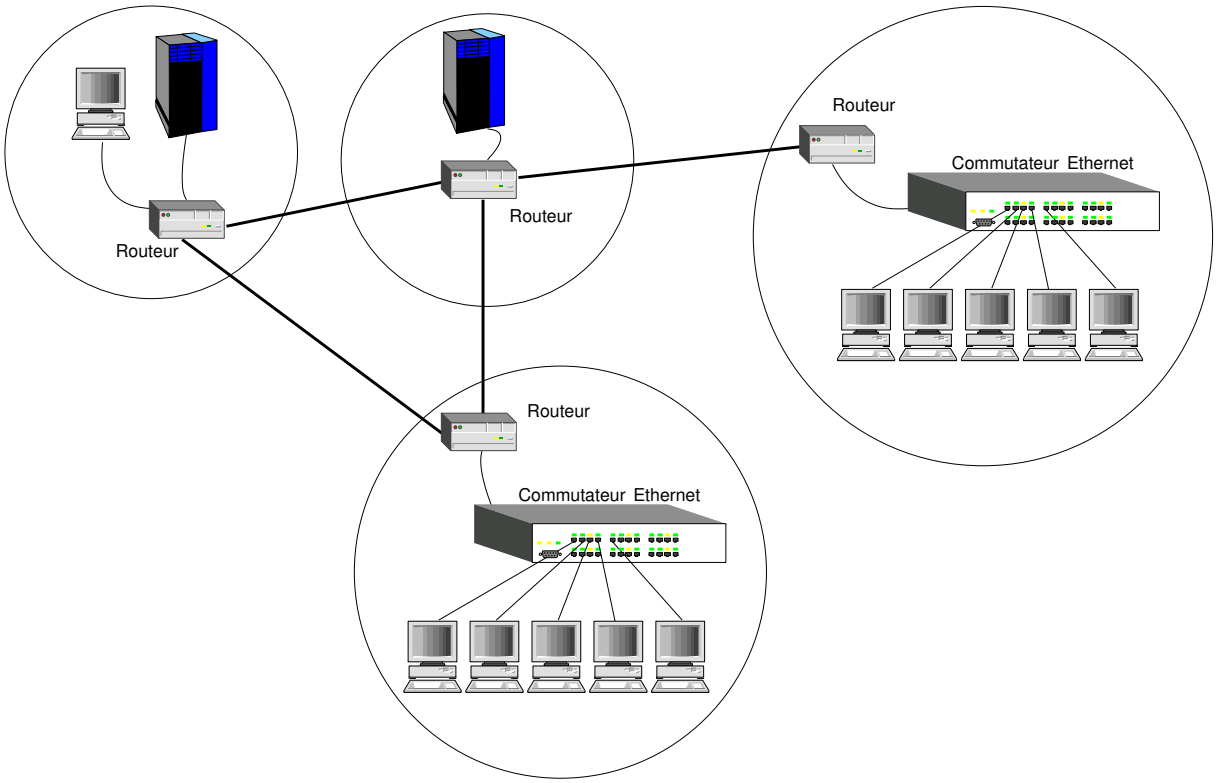

FIG. 8.1 – Architecture simplifiée d'une grille

couche supérieure requiert des informations permettant de savoir sur quelle machine trouver quelle donnée : nous décrirons les besoins et les solutions de l'implantation actuelle par la suite.

Si nous replaçons cela dans le cadre du projet RNTL E-Toile qui utilise l'infrastructure du projet INRIA VTHD, nous avons une telle architecture. Pour informations, plusieurs sites de l'INRIA sont reliés par des liaisons dédiées à Vraiment Très Haut-Débit (10Gb/s), mais les routeurs actuellement en place permettent seulement à ces sites d'utiliser une bande passante d'1 ou 2 Gb/s.

La première version du prototype développée vise à effectuer le transfert de fichiers de manière optimisée entre deux sites utilisant NFSP. La couche supérieure gérant les mises en relation des deux sites (GXFER [DGL03]) a été développée par Christian Guinet pour le compte du projet RNTL E-Toile de manière à pouvoir s'intégrer ultérieurement aux divers composants que ce projet fournira (composants de sécurité et systèmes d'informations principalement).

8.1.2 Considérations d'implantation

Lors d'un transfert grappe à grappe, c'est-à-dire par le biais d'un WAN, l'utilisation d'un protocole tel qu'UDP n'est pas vraiment envisageable pour plusieurs raisons.

Une de ces raisons est qu'en cas d'altération d'un morceau d'un paquet IP dans lequel le datagramme aurait été fragmenté, tout le datagramme va être jeté pour être ré-émis. Ce qui peut avoir un effet limité sur un LAN devient problématique sur un WAN à cause de la latence plus élevée. Avec un protocole tel que TCP, seul le paquet IP manquant doit être ré-émis, ce qui permet globalement d'avoir de meilleures performances.

Une autre raison plus pratique est que l'UDP est peu sûr (il peut être facilement *spoofé*). De plus la gestion des « pseudo-connexions » sur un système pare-feu est relativement peu évoluée, dans la mesure où après émission d'un paquet UDP, le pare-feu doit laisser le port utilisé pour l'émission ouvert pendant une certaine durée pour recevoir un éventuel paquet retour.

De plus, UDP n'assure pas de gestion de la congestion et c'est à l'application d'assurer cela par le biais de *timeouts*, de ré-émissions ou d'évaluation des pertes de messages, ce qui a quelque peu tendance à faire ressembler la nouvelle couche ainsi construite juste au dessus d'UDP à du TCP, donc plutôt que de dupliquer l'effort de développement, utiliser du TCP semble un choix plus que logique.

On peut toutefois remarquer que certains systèmes P2P utilisent de l'UDP mais c'est principalement pour demander des informations à de nombreux pairs car le coût d'envoi d'un message par UDP s'avère très inférieur à un envoi par TCP qui requiert plus de temps (mise en place de la connexion) et de ressources (maintien de la connexion). Cette approche s'avère dans ce cas là intéressante car l'influence de la perte de paquets n'a pas de conséquences graves, si ce n'est de ne pas trouver l'objet recherché (même s'il existe). L'utilisateur peut alors ré-émettre ses requêtes s'il le désire.

Pour revenir au système NFSP, comme nous n'avons pas souhaité que la couche de transfert site-à-site reste trop imbriquée avec le fonctionnement interne de NFSP et donc puisse aussi être utilisable avec d'autres systèmes de fichiers, il a été décidé de ne pas alourdir le comportement des iods NFSP en créant des iods GXFER.

Ces iods GXFER, doivent, à cause de l'architecture hétérogène des sites E-Toile (tous les sites n'ont pas opté pour des grappes Beowulf), pouvoir s'adapter au système de stockage utilisé sur le site en question. Dans le cas d'un site NFSP, les nœuds de stockage auront donc deux serveurs de données : un iod NFSP qui sert à servir localement au sein du LAN les données demandées et qui doit donc rester très léger pour pouvoir offrir une latence très faible à la gestion des diverses requêtes et un iod GXFER qui lui va prendre soin de gérer les transferts inter-LAN et qui par conséquent nécessite moins d'offrir une latence faible, puisqu'il va uniquement dialoguer par le biais de canaux TCP.

Ainsi, ce choix d'architecture conçu comme très modulaire devrait rendre possible sans trop de difficultés l'écriture de couches d'interfaçage avec d'autres systèmes de fichiers parallèles ou avec des bibliothèques d'accès parallèles purs, c'est-à-dire n'offrant pas d'autre interface que celle du VFS pour les applications des utilisateurs.

C'est ainsi qu'au terme de son développement, GXFER devra permettre de pouvoir gérer les transferts d'une grappe vers un serveur centralisé ou entre deux serveurs centralisés, où toute autre combinaison de transfert site-à-site qui apparaîtra dans l'architecture actuellement mise en place dans le projet E-Toile, ces transferts étant à chaque fois le plus optimisés possible, en fonction des spécificités des différents sites.

8.1.3 L'implantation

Nous allons présenter les divers problèmes d'implantations qui se sont posés lors de l'élaboration du prototype et présenter l'interface de communication qu'utilise GXFER avec NFSP.

8.1.3.1 Interface d'accès aux *internals* NFSP

Nous avons défini une API permettant à GXFER d'accéder aux *internals* de NFSP : en effet afin de gérer de manière optimale les transferts sans avoir des coûts supplémentaires de gestion exorbitants, il s'est avéré nécessaire de définir une API minimale d'interfaçage avec le serveur.

GXFER a besoin de plusieurs informations pour pouvoir accéder aux fichiers et doit pouvoir accéder physiquement aux méta-données. À partir de là, il pourra avoir en fonction du nom du fichier et des spécificités de la requête (*offset*, taille) les endroits (machine, nom des fichiers et *offsets*) sur lesquels il pourra accéder aux données qu'il souhaite (en lecture ou en écriture). L'utilisation d'une bibliothèque à lier à l'application nous a paru plus performant que l'ajout d'un service au méta-serveur NFSP, ce

qui certes aurait limité la « dispersion » des algorithmes dans plusieurs modules mais aurait eu un surcoût non-négligeable en accès. De plus, cela aurait pu poser des problèmes avec l'intégration de plusieurs autres services si l'on devait implanter ce service sur d'autres systèmes parallélisés.

La bibliothèque fournie permet de parcourir un fichier de configuration pour récupérer la liste des iods mais elle peut choisir de les récupérer elle-même par tout autre moyen lui convenant. Une fois cette liste obtenue, la fonction principale de la bibliothèque consiste à fournir à l'application l'utilisant, une réponse à la question « comment accéder aux données utiles ? »

Pour ce faire la fonctionnalité principale est la fonction d'accès aux paramètre de *striping* :

```
struct nfsp_iov_t {
  int iodno;           // numéro de l'iod
  char *localfn;       // nom local des données sur l'iod
  off_t wanted_offset; // offset logique demandé par l'application
  off_t mapped_offset; // offset physique auquel l'application accèdera
  size_t size;         // taille de l'accès
};
int nfsp_get_iov_from_filename(struct nfsp_iov_t **ppiov, int *piov_size,
        const char * const filein, const off_t offset, const size_t size);
```

À partir du nom d'un méta-fichier, et d'une zone de données contiguës à lire, elle fournit en retour un vecteur indiquant où se trouvent les données (numéro de la machine, nom du fichier de données ainsi que les adresses où les données peuvent être lues).

Par exemple supposons que le système contienne 2 iods numérotés 0 et 1 et un fichier « `toto` » d'inode 0x42424242 et de *seed* 0x12345678 de taille 256KB (avec une taille de blocs de 64KB). L'application effectue donc un appel pour savoir comment accéder aux données de « `toto` » comprises entre 42 et 135KB. En retour, celle-ci va recevoir les informations suivants :

```
piov_size=3
ppiov[0]={0, « i42424242s12345678 », 42K, 42K, 64-42 = 22K}
ppiov[1]={1, « i42424242s12345678 », 64K, 0K, 64K}
ppiov[2]={0, « i42424242s12345678 », 128K, 64K, 135-128 = 7K}
```

Ceci signifie que l'application doit accéder au fichier `i42424242s12345678` sur l'iod 0 et qu'il peut lire 22K a partir de l'offset 42K du fichier. De même, la zone de données de « toto » entre 64K et 128K se trouve entre les offsets 0 et 64K du fichier `i42424242s12345678` sur l'iod 1.

8.1.3.2 Transferts entre deux sites utilisant NFSP

Dans le cas de tels transferts, des problèmes apparaissent lorsque le nombre de serveurs de stockage diffère d'un site à un autre car il faut prendre en compte le fait que

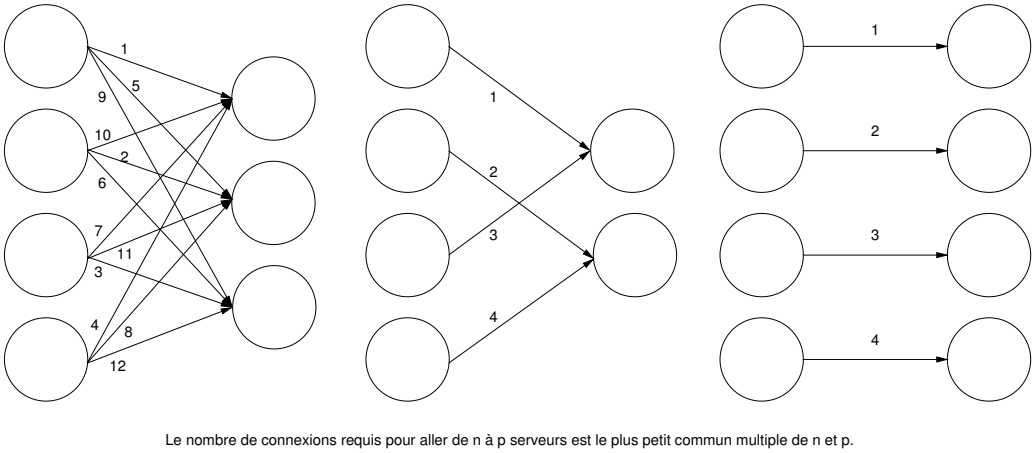

FIG. 8.2 – Connexions à établir quand le nombre de serveurs de stockage diffère

chaque serveur peut avoir à maintenir une connexion avec plusieurs sites. Le nombre minimal de connexions à établir et mettre en place est illustré sur la figure 8.2.

Il est possible d'envisager de multiplier les connexions entre deux nœuds distants afin d'essayer d'avoir plus de performances mais cette approche risque de produire l'effet inverse car les machines risquent d'être plus stressées.

Le nombre de connexions total à maintenir dans ce cas là correspond au plus petit commun multiple des nombres d'iods. À ce nombre s'ajoute une connexion utilisée pour le contrôle et la mise en relation des deux serveurs GXFER.

De plus, dans le cas où le nombre d'iods diminue, il faut aussi faire attention à ne pas trop surcharger un des iods, quitte à ralentir un peu le transfert pour éviter la contention des entrées/sorties sur le système de l'iod.

D'une manière simplifiée le déroulement d'une opération de récupération d'un fichier distant (`GET`) se déroule en plusieurs étapes qui consistent en une phase de « négociations » puis une phase de transferts :

1. trouver le méta-serveur gérant le fichier
2. récupérer les méta-données afin de pouvoir calculer la localisation des données le constituant
3. créer la méta-donnée en local
4. ouvrir les connexions avec les iods locaux et leur fournir les paramètres du transfert (les deux méta-données locale et distante)
5. chaque iod local calcule alors les transferts qui vont devoir être effectués et les iods distants qui vont entrer en jeu
6. établissement des connexions iods locaux - iods distants
7. transferts

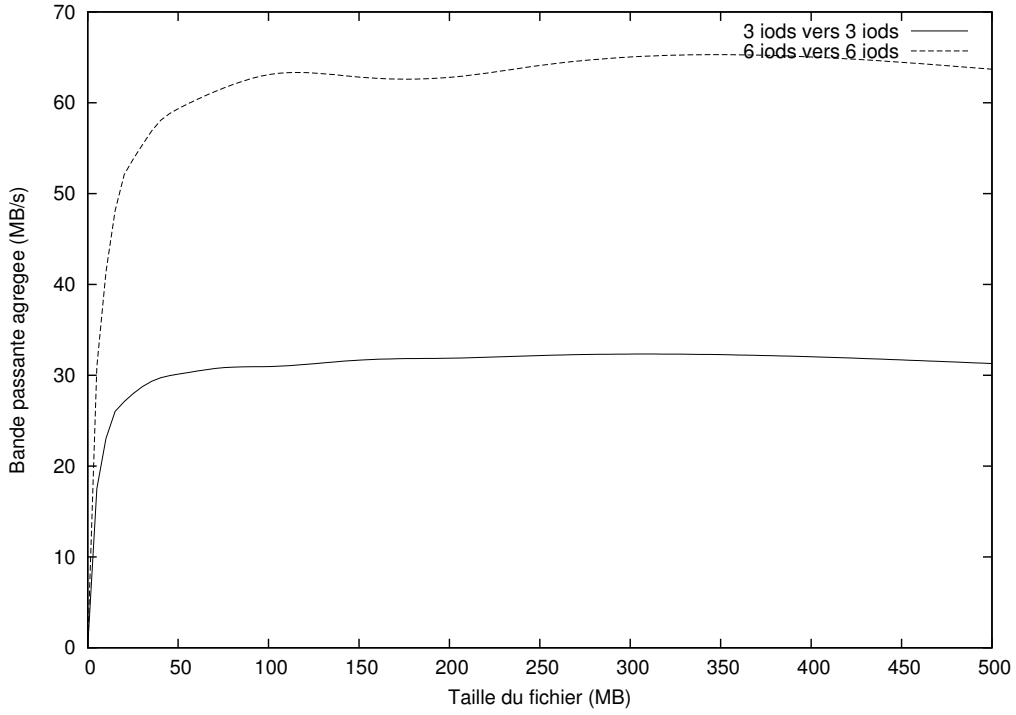

FIG. 8.3 – Performances de GXFER avec 3x3 et 6x6 iods

8. quand le serveur aura reçu un acquittement de chaque iod (c'est-à-dire que tous les transferts des morceaux seront finis), le fichier est transféré

Le prototype GXFER fonctionne actuellement uniquement avec une couche de stockage NFSP mais dans la mesure où nous avons souhaité que les deux composants puissent être facilement adaptables, tout autre système de stockage peut être envisagé tel qu'un système de fichier local (EXT2, EXT3, ...) ou bien un autre système de fichiers par réseau.

8.1.4 Évaluation

Les premières mesures (voir figure 8.3) réalisées par Christian Guinet entre l'*i-cluster* (INRIA-Grenoble) et l'UREC de Lyon (environ 100km) se sont montrées très prometteuses, les taux de transferts étant quasiment proches de l'optimum. Ces performances sont celles obtenues avec des machines en configuration standard (pas de *tuning* particulier des machines au niveau de la configuration réseau ou disques).

Une des remarques que l'on peut faire sur ces courbes est le coût de l'établissement des connexions : sur les petits fichiers, la mécanique mise en œuvre peut paraître lourde mais elle devient vite rentabilisée lorsque les fichiers à transférer commencent à devenir conséquents, mais c'est le cas dans de nombreux protocoles de transferts. Actuellement le prototype fonctionne en mode fichier-par-fichier mais une solution en-

visageable peut être de *pipeliner* les transferts d'une manière similaire à ce que fait HTTP/1.1 et ainsi d'éviter les coûts de re-établissement de connexions.

Étant actuellement relié avec Lyon par un lien gigabit (125MB/s comme maximum théorique), des mesures que nous avons effectuées avec 10 machines de part et d'autres nous ont permis d'obtenir en novembre 2003, plus de 100MB/s (presque 110MB/s) de vitesse de transfert, ce qui nous a permis d'être confortés dans les différents choix architecturaux ayant été effectués.

D'autres expérimentations restent encore à faire avec un nombre plus grand d'iods de part et d'autre, et avec des serveurs ayant une latence entre eux plus importante. Nous avons de bons espoirs quant aux tests et expérimentations qui pourront être réalisées dans le cadre du projet RNTL E-toile. Une intégration avec la brique QoSinus [CP03] est aussi en cours d'étude, afin de pouvoir avoir de la qualité de service pour effectuer des transferts à grande vitesse.

8.2 Application de système de fichiers distribués à plus grande échelle

En conservant en mémoire l'omniprésence du protocole NFS, et toujours en utilisant une séparation de la gestion des données et méta-données, Olivier Valentin a commencé à mettre au point un prototype de système de fichiers globalisé pour grilles, NFSG [Val03] [LLG+03].

De même que dans le domaine du stockage, les approches dans ce domaine sont menées depuis de nombreuses années avec des logiciels tels que AFS, CODA / SlashGrid, les premières versions de xFS [WA93], Pangaea [SK02] [SKKM02] plus récemment.

Comme dans NFSP et GXFER, le principe est de séparer la gestion des données et des méta-données en gérant le tout avec une cohérence temporelle. De plus, l'un des buts de ce système est d'offrir un espace de nommage global au sein d'une grappe de grappes.

D'une manière similaire à ce qui se faisait dans Sprite (cf. section 4.3.2 page 61), chaque serveur NFSG va gérer une sous partie de l'arborescence d'un arbre global. Une différence de l'approche retenue est que chaque site/serveur ne va pas forcément contenir les données associées à la partie d'arbre qu'il gère, l'idée étant que les données - qui peuvent être situées ailleurs - soient rapatriées à grandes vitesse depuis leur site de stockage par exemple le couple NFSP/GXFER. Le principe de cette répartition est illustré dans la figure 8.4.

Le système doit démarrer de façon progressive, toutes les sites s'enregistrant au fur et à mesure auprès du serveur maître. Par la suite, si une méta-donnée vient à manquer sur un site, une vérification sur le maître de sa localisation va avoir lieu. À partir de ce moment, le transfert de la méta-donnée peut être effectué, puis éventuellement les

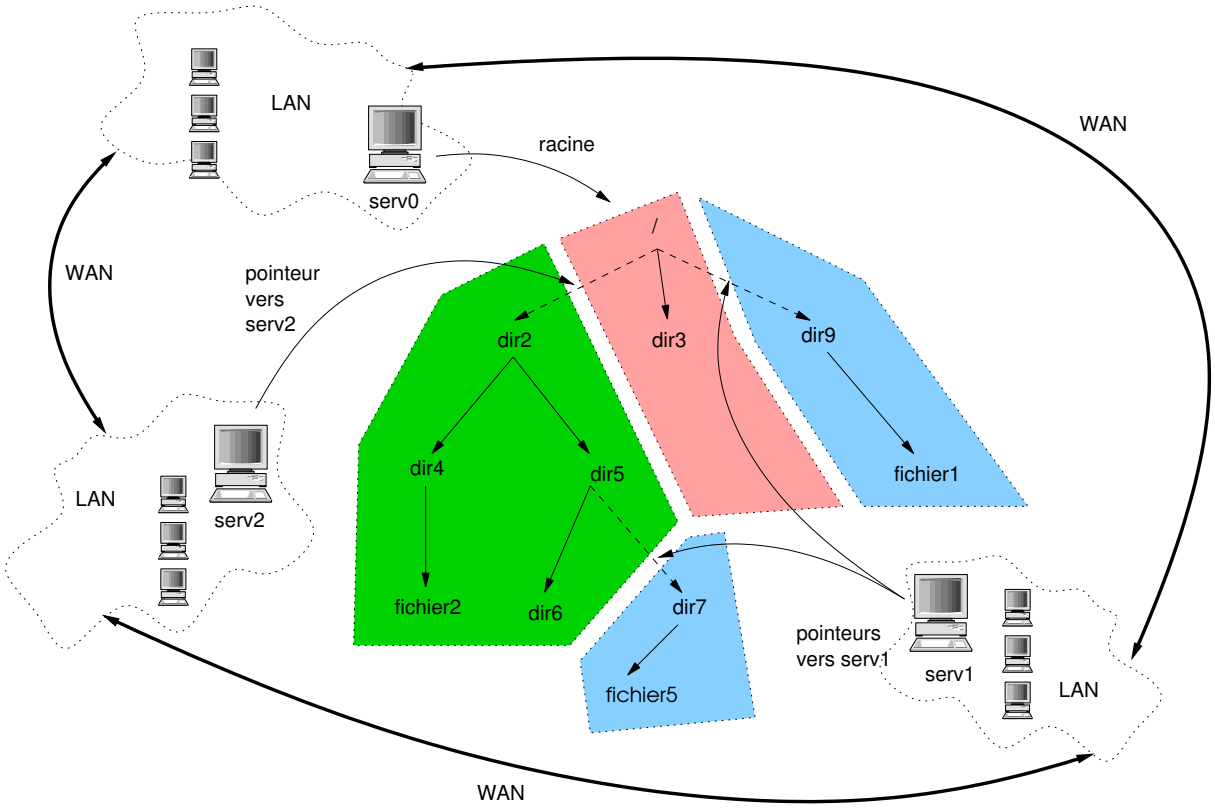

FIG. 8.4 – Décomposition arborescente de NFSg et lien avec un WAN

données (si le mode choisi est un mode de pré-chargement) ou bien au premier accès (si c'est un mode d'accès « à la volée »).

La cohérence offerte par un tel modèle est relâchée puisque très proche de celle offerte par NFS. L'avantage de cette approche est une transparence sur les clients puisque ces derniers ne voient que leur serveur NFSg de grappe donc local.

Le développement de ce modèle est encore en cours et demandera quelques modifications à NFSP pour pouvoir s'intégrer de manière optimale dans ce modèle avec GXFER et ainsi fournir des performances optimales.

8.3 Conclusion

Nous venons donc de décrire dans ce chapitre quelques applications développées autour de notre outil de base NFSP. Ces applications visent pour la plupart une cible de matériel non dédié et utilisent aussi une gestion relâchée de la gestion des données et des méta-données.

GXFER permet ainsi de fournir un service de transfert performant entre deux grappes reliées par une liaison à haut débit. Les travaux actuels concernant GXFER sont principalement axés sur le support d'autres couches de stockage ainsi qu'une normalisation du protocole[DGL03].

NFSg fournit quant à lui un service de système de fichiers distribués sur plusieurs grappes tout en s'interfaçant avec NFSP et NFSg pour tirer partie de l'architecture de grappes de grappes. Le prototype requiert encore une certaine quantité de développement pour fonctionner au mieux, mais les grandes lignes de l'implantation en sont tracées.

Chapitre 9

Conclusions et perspectives

Ce chapitre a pour but d'effectuer un résumé des divers sujets abordés dans ce document. Nous donnerons ensuite nos conclusions sur ceux-ci puis nous envisagerons diverses extensions qui pourront constituer la base de nouvelles expérimentations futures.

9.1 Conclusions

Dans une première partie (chapitre 2 page 21), nous avons tenté de mettre en avant les différents problèmes liés à la conception d'un système de stockage, notamment la différence de traitement des données et des méta-données. Nous avons ensuite décrit dans le chapitre 3 page 31 le fonctionnement de divers moyens et modèles existants pour accéder aux données (bibliothèques, outils de transferts, ...)

Par la suite, nous nous sommes focalisés sur l'un de ces moyens, à savoir les systèmes de fichiers dans le chapitre 4 page 43. Nous avons souhaité effectuer une présentation de quelques systèmes de fichiers distribués qui nous ont paru intéressants afin d'observer les différences d'approches qui peuvent être utilisées dans un cadre d'accès distribué avec l'utilisation de modèles à disques partagées et d'autres (la majorité) à messages. Comme un point important était d'avoir un système standard, nous nous sommes donc concentrés sur des travaux autour du système le plus utilisé, NFS.

Le chapitre suivant (chapitre 5 page 71) a souhaité précisé le cadre de ces travaux ainsi que divers autres travaux réalisés autour des serveurs NFS. De plus, nous rappelons quelques unes des contraintes que nous avons dû respecter, notamment un respect des standards, un système facilement modifiable et intégrable en tant que brique de base pour d'autres logiciels et un fonctionnement efficace sur des machines « simples » peu chères (*commodity hardware*).

Dans le chapitre 6 page 83, nous avons procédé à une description des implantations réalisées (mode utilisateur et noyau) ainsi qu'à une évaluation de leurs performances. Ces implantations ont par la suite subi plusieurs expérimentations (décrites dans le

chapitre 7 page 111) afin d'en améliorer les performances grâce à la multiplication des points d'entrées ou bien en améliorant la sûreté du stockage avec la mise en place de redondance.

Nous avons ensuite décrit dans un dernier chapitre (chapitre 8 page 125) une expérience d'utilisation de notre prototype comme brique de base dans le cadre de transferts performants à haut débit entre deux sites et dans le cadre d'une utilisation pour la réalisation d'un systèmes de fichiers globalisé offrant une cohérence de type NFS.

9.2 Bilan

Les travaux que nous avons été amenés à réaliser ont vu un de leurs objectifs atteints notamment en proposant une brique de base respectant nos objectifs : en utilisant un matériel peu onéreux et absolument pas spécialisé, nous avons pu mettre en place un système performant et simple à installer (car ayant très peu de différences avec un serveur NFS normal) pour servir des fichiers sur une grappe ou un réseau local (LAN). Ce système permet aussi à l'application développée au dessus de fournir un service de transfert rapide de grappe à grappe et ce, de manière optimisée et sans avoir recours à du matériel spécialisé.

L'utilisation d'un protocole « figé » s'est avéré très contraignante car à chaque modification du prototype, la question « est-ce que ça ne va rien casser ? » (ou bien en version allégée : « est-ce que cette modification ne va rien *trop* casser ? ») devait être posée.

La contrepartie de ce désavantage est que cela permet aussi, à notre avis, de lutter contre la lente pénétration « sur le marché » de nouveaux systèmes plus performants mais très différents. Par exemple, AFS est développé depuis bientôt 20 ans et, malgré ses qualités indéniables, il ne reste que peu utilisé, la plupart des sites préférant utiliser des systèmes intégrés (NAS) offrant du NFS et du CIFS. Les raisons de cette lente migration viennent très probablement des facteurs suivants : la base établie est énorme et les délais de migrations de infrastructures matérielles et réseau sont souvent très élevés, les systèmes sont bien testés et ont donc de bonnes raisons d'être plus stables et de demander moins de maintenance.

Un autre exemple de ces observations peut être vu dans la mise en place de systèmes utilisant NFS4 : bien que corrigeant la plupart des défauts des versions précédentes, très peu de systèmes incluent en standard le support du client (encore moins le support serveur). Le futur noyau Linux 2.6 devrait inclure ces extensions mais il risque d'être bien seul dans le paysage informatique[1].

[1] À moins que les systèmes Linux n'atteignent la domination totale que Linus Torvalds envisageait 8-)

9.3 Perspectives

Comme la plupart des implantations de prototypes, un travail de remise à plat des concepts utilisés voire une ré-implantation « propre » en utilisant des concepts développés par d'autres projets peut s'avérer être un fil vers de nouvelles approches. Par exemple, Cuckoo et Slice jouent le rôle de *proxies* allégés et les serveurs apparaissent donc beaucoup moins stressés. Aussi, un allègement du méta-serveur (en le faisant passer de la gestion de la couche RPC à une couche UDP) pourrait-il constituer un aspect intéressant pour de nouveaux travaux. Passer à des systèmes fondés sur des RPC sur TCP peut, malgré de plus grandes contraintes, aussi s'avérer intéressant pour plusieurs raisons : gestion intégrée de la contention réseau, support WAN, interactions avec les extensions futures du protocole NFS4, ...

D'autres pistes d'études semblent se dessiner pour le futur de notre solution de stockage :

- La multiplication des points d'entrée est un point fondamental du système pour pouvoir supporter de grosses grappes Beowulf afin d'offrir un meilleur passage à l'échelle. Les premiers résultats de la thèse de Rafael Ávila obtenus avec une version « naïve » de la gestion des fautes de méta-fichiers [ÁLL+] semblent prometteurs mais requièrent encore des tests plus poussés afin de s'assurer de la validité de l'approche.
- Un point adjacent à la multiplication des points d'entrée peut à notre avis consister en la mise en place d'une hiérarchie de serveurs de méta-données. Nous nous sommes limités à un niveau de hiérarchie dans les différentes approches que nous avons évoquées, mais en utilisant un arbre, ou des techniques de diffusion plus évoluées sur des arbres couvrants comme celles développées dans la thèse de Cyrille Martin [Mar03], l'extensibilité de la gestion des méta-données a de bonnes chances de pouvoir être atteinte. Cependant, l'utilisation de NFS pose le problème de ne pas trop faire croître la latence des opérations sous peine de générer des *timeouts*, ou de diminuer les performances du système.
- La gestion de modèles plus complexes de distributions de données doit aussi être évaluée. Le mode actuel est un mode simple (les fichiers sont distribués par blocs de manière cyclique) car, comme nous venons de le dire, le méta-serveur doit avoir une latence faible sinon les clients risquent de ré-émettre leurs requêtes. Ceci implique que les opérations à réaliser doivent être rapides et, donc, simples à effectuer. Des modèles de réplications utilisant des traces des accès précédents pour optimiser le placement et mettre en place des techniques de type réplication de données peuvent être envisagées, si elles ne s'avèrent pas trop coûteuses à déployer (ou alors elles peuvent être effectuées dans les moments où le système est peu chargé). Il faut aussi veiller à ce que les méta-données ne grossissent pas trop afin de pouvoir rester facilement dans les caches des différentes entités amenées à les manipuler, ce qui peut aussi limiter les modèles applicables.
- Un corollaire du point précédent peut être vu dans une gestion simple de la reconfiguration dynamique : l'ajout de nœuds peut être géré sans trop de problèmes (en augmentant la taille du *stripe* pour les nouveaux fichiers) mais des techniques

plus poussées avec un démon de nettoyage à la Zebra ou xFS pourraient aussi être intéressantes.
– NFSP gérant le *striping* des données, l'implantation d'une API compatible MPI-IO peut aussi être un supplément fonctionnel qui peut rendre service à certains types d'applications (la plupart des implantations MPI/IO gérant elles mêmes les verrous et autres synchronisations). La thèse qu'a récemment commencée Adrien Lebre au sein du laboratoire ID-IMAG avec la collaboration de Bull sur ce sujet consistera, entre autres buts, à mener une étude des modèles d'E/S parallèles applicables à ce support.
– Des tests avec plus d'applications et une charge de travail moins « synthétique » devraient permettre de mieux cerner les limites du fonctionnement des divers prototypes.

Avec l'arrivée de nouvelles grappes ayant un rapport performance/prix de plus en plus intéressant et avec des grappes d'architectures différentes (grappes de bi-Itanium II et grappes de bi-Xéon), il sera aussi intéressant d'observer le comportement de notre brique sur ces systèmes. Ces systèmes étant constitués de machines optimisées pour les grappes, il sera aussi intéressant de voir si notre solution s'adapte bien à ces nouvelles architectures plus performantes. De plus, dans le cadre de l'évolution des réseaux WAN et avec l'arrivée de routeurs plus performants, il faudra étudier dans quelle mesure les transferts de données pourront être optimisés avec les outils développés.

En revanche, un tel aspect peut s'avérer intéressant dans le cadre des développements effectués autour des concepts grappes de calculs sur machines de bureautique (donc moins puissantes que des machines dédiées conçues pour fonctionner en grappes de calcul). L'utilisation de ces jachères de calcul n'est pas une nouveauté car, déjà depuis un moment, avec Condor,par exemple, (mais aussi plus récemment avec la thèse de Bruno Richard [Ric03]), la chasse à cette puissance d'appoint commençait à susciter un vif intérêt chez de nombreux scientifiques… En effet, les généticiens et les physiciens ont souvent de gros besoins en puissance de calcul, mais aussi en stockage de données. Les solutions ici proposées concernant le stockage, conjuguées à un système de gestion des jachères, pourraient ainsi constituer une possibilité d'offrir de la puissance de calcul, mais aussi du stockage à moindre coût pour les utilisateurs les plus demandeurs de ces ressources.

Bibliographie

[ABB+01] W. ALLCOCK, J. BESTER, J. BRESNAHAN, A. CHERVENAK, L. LIMING et S. TUECKE. « GridFTP : Protocol Extensions to FTP for the Grid », mars 2001. Expiration en août 2001.

[ABDM01] Philippe AUGERAT, Wilfrid BILLOT, Simon DERR et Cyrille MARTIN. « A Scalable File Distribution and Operating System Installation Toolkit for Clusters ». Rapport de recherche, ID-IMAG, octobre 2001. Disponible à `http://ka-tools.sourceforge.net/publications/file-distribution.pdf` (2003-09-22).

[ABM+02] S. ATCHLEY, M. BECK, J. MILLAR, T. MOORE, J. S. PLANK et S. SOLTESZ. « The Logistical Networking Testbed ». Rapport Technique CS-02-496, University of Tennessee, décembre 2002. Disponible à `http://www.cs.utk.edu/~plank/plank/papers/CS-02-496.html` (2003-08-13).

[AC02] Darrell ANDERSON et Jeff CHASE. « Failure-Atomic File Access in the Slice Interposed Network Storage System ». *IEEE Journal on Cluster Computing*, 5(4), 2002.

[ACV00] D. ANDERSON, J. CHASE et A. VAHDAT. « Interposed request routing for scalable network storage », 2000. Disponible à `http://www.cs.duke.edu/ari/publications/slice-osdi2000/` (2003-08-11).

[ADM01] Philippe AUGERAT, Simon DERR et Stéphane MARTIN. « Outils d'exploitation de grappe de PC ». Dans *Actes des Journées Réseaux 2001 (JRES 2001)*, Lyon, France, novembre 2001.

[ADN+95] Thomas ANDERSON, Michael DAHLIN, Jeanna NEEFE, David PATTERSON, Drew ROSELLI et Randolph WANG. « Serverless network file systems ». Dans ACM, éditeur, *Proceedings of the 15th Symposium on Operating System Principles*, pages 109–126, décembre 1995.

[ASDZ00] Mohit ARON, Darren SANDERS, Peter DRUSCHEL et Willy ZWAENEPOEL. « Scalable Content-Aware Request Distribution in Cluster-based Network Servers ». Dans *Proceedings of the 2000 Annual Usenix Technical Conference*, pages 323–336, San Diego, CA, USA, juin 2000.

[BBMP02] A. BASSI, M. BECK, T. MOORE et J. PLANK. « The Logistical Backbone : Scalable Infrastructure for Global Data Grids ». Dans *Proceedings*

of Asian Computing Science Conference 2002, Hanoi, Vietnam, décembre 2002. Springer-Verlag.

[BEM91] Anupam BHIDE, E. N. ELNOZAHY et Stephen P. MORGAN. « A Highly Available Network File Server ». Dans *Proceedings of the USENIX Winter 1991 Technical Conference*, pages 199–206, 1991.

[BFK+99] Joseph BESTER, Ian FOSTER, Carl KESSELMAN, Jean TEDESCO et Steven TUECKE. « GASS : A Data Movement and Access Service for Wide Area Computing Systems ». Dans *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 78–88. ACM Press, 1999.

[BH99] Scott BAKER et John H. HARTMAN. « The Gecko NFS Web proxy ». Dans *Computer Networks*, volume 31, pages 1725–1736, Tucson, USA, 1999.

[BH02] Scott BAKER et John H. HARTMAN. « The Mirage NFS Router ». Technical Report TR02-04, University of Arizona, novembre 2002. Disponible à `http://www.cs.arizona.edu/research/reports.html` (2003-08-11).

[BLFF95] T. BERNERS-LEE, R. FIELDING et H. FRYSTYK. « Hypertext Transfer Protocol – HTTP/1.0 ». RFC1945, mai 1995.

[BN99] Peter J. BRAAM et Philip A. NELSON. « Removing Bottlenecks in Distributed Filesystems : Coda and Intermezzo as examples ». *Carnegie Mellon University and Western Washington University*, 1999.

[Bra03] Peter J. BRAAM. « The LUSTRE Storage Architecture », mars 2003. Disponible à `http://www.lustre.org/docs/lustre.pdf` (2003-08-21).

[BZ01] Peter J. BRAAM et Rumi ZAHIR. « Lustre Technical Project Summary (Attachment A to RFP B514193 Response) ». Rapport Technique, Cluster File Systems, Inc., juillet 2001. Disponible à `http://www.lustre.org/docs/lustre-sow-dist.pdf` (2003-08-20).

[CDM97] Rémy CARD, Eric DUMAS et Franck MÉVEL. *Programmation Linux 2.0 : API système et fonctionnement du noyau*. Eyrolles, 1997.

[Cec02] Emmanuel CECCHET. « Whoops ! : a Clustered Web Cache for DSM Systems ». Dans *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02)*, Vienne, Autriche, juillet 2002.

[CER96] CERT. « CA-1996-21 : TCP SYN Flooding and IP Spoofing Attacks ». `http://www.cert.org/advisories/CA-1996-21.html`, septembre 1996.

[CF96] Peter F. CORBETT et Dror G. FEITELSON. « The Vesta parallel file system ». Dans *ACM Transactions on Computer Systems (TOCS)*, pages 225–264, août 1996. Disponible à `http://portal.acm.org/citation.cfm?id=233558` (2003-07-22).

[CFS02] Inc. CLUSTER FILE SYSTEMS. « LUSTRE : A High-Performance, Scalable, Open Distributed File System for Clusters and Shared-Data Environments », novembre 2002. Disponible à `http://www.lustre.org/docs/whitepaper.pdf` (2003-08-21).

[CGC+02] Alejandro CALDERÓN, Félix GARCÍA, Jesús CARRETERO, Jose M. PÉREZ et Javier FERNÁNDEZ. « An Implementation of MPI-IO on Expand : A Parallel File System Based on NFS Servers ». Dans *9th PVM/MPI European User's Group*, septembre 2002. Disponible à `http://link.springer.de/link/service/series/0558/bibs/2474/24740306.htm` (2003-08-21).

[CIRT00] Philip H. CARNS, Walter B. Ligon III, Robert B. ROSS et Rajeev THAKUR. « PVFS : A Parallel File System for Linux Clusters ». Dans *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association.

[Coh] Bram COHEN. « Fonctionnement et description du protocole BitTorrent ». Disponible à `http://bitconjurer.org/BitTorrent/`.

[Coz03] Olivier COZETTE. « READ2 : Partager les disques au niveau réseau ». Dans *Proceedings de RENPAR'15/CFSE'3/SympAAA'2003*, octobre 2003.

[CP03] Fabien CHANUSSOT et Pascale PRIMET. « Spécification du service actif QoSINUS ». Rapport de Recherche RT-0287, INRIA, octobre 2003.

[CPS95] B. CALLAGHAN, B. PAWLOWSKI et P. STAUBACH. « NFS Version 3 Protocol Specification ». RFC1813, juin 1995.

[CRT+00] B. CALLAGHAN, D. ROBINSON, R. THURLOW, C. BEAME, M. EISLER et D. NOVECK. « NFS Version 4 Protocol ». RFC3010, novembre 2000.

[DGL03] Yves DENNEULIN, Christian GUINET et Pierre LOMBARD. « A Parallel Data Transfer Solution for Grids ». Dans *Présentation effectuée au Global Grid Forum (GGF8)*, Seattle, WA, USA, juin 2003.

[DKK+01] Frank DABEK, M. Frans KAASHOEK, David KARGER, Robert MORRIS et Ion STOICA. « Wide-area cooperative storage with CFS ». Dans *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, octobre 2001.

[FGM+99] R. FIELDING, J. GETTYS, J. MOGUL, H. FRYSTYK, L. MASINTER, P. LEACH et T. BERNERS-LEE. « Hypertext Transfer Protocol – HTTP/1.1 ». RFC2616, juin 1999.

[FK99] Ian FOSTER et Carl KESSELMAN, éditeurs. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[FLP85] Michael J. FISCHER, Nancy A. LYNCH et Michael S. PATERSON. « Impossibility of Distributed Consensus with One Faulty Process ». *Journal of the ACM (JACM)*, 31(2), avril 1985.

[Fyo] FYODOR. « nmap (Site Web) ». `http://www.insecure.org/nmap/`.

[Gog03] Brice GOGLIN. « ORFA, Optimizing Access to Remote File Systems », juillet 2003. Présentation disponible à `http://www.ens-lyon.fr/~bgoglin/` (2003-07-22).

[GS97] Rachid GUERRAOUI et André SCHIPER. « Software-Based Replication for Fault Tolerance ». *Computer*, 30(4) :68–74, 1997.

[GS02] Deepak GUPTA et Vikrant SHARMA. « Design and Implementation of a Portable and Extensible FTP to NFS Gateway ». Dans *Proceedings of Principles and Practice of Programming in Java (PPPJ'02)*, Dublin. Irelande, juin 2002.

[HE98] P. HETHMON et R. ELZ. « Feature negotiation for the File Transfer Protocol ». RFC2389, août 1998.

[HGFW02] Harry HULEN, Otis GRAF, Keith FITZGERALD et Richard W. WATSON. « Storage Area Networks and the High Performance Storage System ». Dans *Tenth NASA Goddard Conference on Mass Storage Systems*, University of Maryland, avril 2002.

[HL97] M. HOROWITZ et S. LUNT. « FTP Security Extensions ». RFC2228, octobre 1997.

[HL02] Jørgen HANSEN et Renaud LACHAIZE. « Using Idle Disks in a Cluster as a High-Performance Storage System ». Dans *Proceedings of IEEE International Conference on Cluster Computing (Cluster 2002)*, septembre 2002.

[HMS99] John H. HARTMAN, Ian MURDOCK et Tammo SPALINK. « The Swarm Scalable Storage System ». Dans IEEE, éditeur, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*. IEEE, 1999.

[HO01] John H. HARTMAN et John K. OUSTERHOUT. The Zebra Striped Network File System. Dans *High Performance Mass Storage and Parallel I/O : Technologies and Applications*, pages 309–329. IEEE Computer Society Press and Wiley, New York, NY, USA, 2001.

[HS96] R. L. HASKIN et F. B. SCHMUCK. « The Tiger Shark File System ». Dans *Proceedings of the 41st IEEE Computer Society International Conference (COMPCON '96)*, pages 226–231, Santa Clara, CA, USA, 1996.

[IN2] IN2P3. « RFIO (Site Web) ». `http://doc.in2p3.fr/doc/public/products/rfio/rfio.html`.

[KG02] Andrew J. KLOSTERMAN et Gregory GANGER. « Cuckoo : Layered clustering for NFS ». Technical Report CMU-CS-02-183, Carnegie-Mellon University, octobre 2002. Disponible à `http://www.pdl.cmu.edu/PDL-FTP/FS/CMU-CS-02-182_abs.html` (2003-08-11).

[KMM94] Gene H. KIM, Ronald G. MINNICH et Larry MCVOY. « Bigfoot-NFS : A Parallel File-Striping NFS Server », 1994.

[Lab] Lawrence Berkeley LABORATORY. « The Distributed-Parallel Storage System (DPSS) Home Page ». Site web à `http://www-itg.lbl.gov/DPSS/`.

[LD02a] Pierre LOMBARD et Yves DENNEULIN. « nfsp : A Distributed NFS Server for Cluster of Workstations ». Dans *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, avril 2002.

[LD02b] Pierre LOMBARD et Yves DENNEULIN. « Serveur NFS distribué pour grappe de PCs ». Dans *Actes de RenPar'14/ASF/SympA'8*, Hammamet, Tunisie, avril 2002.

[LDVL03] Pierre LOMBARD, Yves DENNEULIN, Olivier VALENTIN et Adrien LEBRE. « Improving the Performances of a Distributed NFS Implementation ». Dans *To appear in the Proceedings of the Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM 2003)*, Lecture Notes in Computer Science. Springer-Verlag, septembre 2003.

[Leb02] Adrien LEBRE. « Composition de service de données et de méta-données dans un système de fichiers distribué ». Rapport de DEA, IMAG, juin 2002. Disponible à `http://www-id.imag.fr/Laboratoire/Membres/Lebre_Adrien/DEA/` (2003-08-21).

[LGG+91] Barbara LISKOV, Sanjay GHEMAWAT, Robert GRUBER, Paul JOHNSON, Liuba SHRIRA et Michael WILLIAMS. « Replication in the Harp File System ». Dans *Proceedings of 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 226–38. Association for Computing Machinery SIGOPS, 1991.

[LLG+03] Pierre LOMBARD, Adrien LEBRE, Christian GUINET, Olivier VALENTIN et Yves DENNEULIN. « Distributed Filesystem for Clusters and Grids ». Dans *Workshop at the Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM 2003), Special session : Large Scale Scientific Computations*, septembre 2003.

[LR99] W. LIGON et R. ROSS. « An Overview of the Parallel Virtual File System ». Dans *Proceedings of the 1999 Extreme Linux Workshop*, juin 1999.

[LT96] Edward K. LEE et Chandramohan A. THEKKATH. « Petal : Distributed Virtual Disks ». Dans ACM, éditeur, *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLO-7)*, octobre 1996.

[Mac] Pavel MACHEK. « POrtable Dodgy Filesystems in Userland (hacK) version 2 ». Site Web à `http://uservfs.sourceforge.net/`.

[Mac94] R. MACKLEM. « Not Quite NFS, Soft Cache Consistency for NFS ». Dans *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 261–278, San Francisco, CA, USA, 1994.

[MAF+02] Kostas MAGOUTIS, Salimah ADDETIA, Alexandra FEDOROVA, Margo I. SELTZER, Jeffrey S. CHASE, Andrew J. GALLATIN, Richard KISLEY, Rajiv G. WICKREMESINGHE et Eran GABBER. « Structure and Performance of the Direct Access File System ». Dans *Proceedings of USENIX : Annual Technical Conference*, 2002.

[Mar03] Cyrille MARTIN. « *Déploiement et contrôle d'applications parallèles sur grappes de grande taille* ». PhD thesis, Institut National Polytechnique de Grenoble, décembre 2003.

[Maz00] David MAZIÈRES. « *Self-certifying file system* ». PhD thesis, MIT, mai 2000.

[MBKQ96] Marshall Kirk MCKUSICK, Keith BOSTIC, Michael KARELS et John QUARTERMAN. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, Inc., 1996. Chapitre 9 (NFS Backgrounder) disponible à `http://www.netapp.com/tech_library/nfsbook.html` (2003-08-18).

[McN02] Andrew MCNAB. « SlashGrid - a framework for Grid aware filesystems », mars 2002. Présentation disponible à `http://www.hep.man.ac.uk/u/mcnab/grid/`.

[MH00] Ian MURDOCK et John H. HARTMAN. « Swarm : A Log-Structured Storage System For Linux ». Dans *Proceedings of FREENIX Track : 2000 USENIX Annual Technical Conference*, juin 2000.

[Mic99] MICROSOFT. « Distributed File System : A Logical View of Physical Storage », avril 1999. Disponible à `http://www.microsoft.com/windows2000/techinfo/howitworks/fileandprint/d%fsnew.asp` (2003-08-21).

[MIT02] MIT. « Lecture 9 : Overlay routing networks (Akarouting) », 2002. Présentation disponible à `http://ocw.mit.edu/OcwWeb/Mathematics/18-996Topics-in-Theoretical-Compu%ter-Science---Internet-Research-ProblemsSpring2002/LectureNotes/index.htm` (2003-08-13).

[MJLF84] Marshall K. MCKUSICK, William N. JOY, Samuel J. LEFFLER et Robert S. FABRY. « A Fast File System for UNIX ». *Computer Systems*, 2(3) :181–197, 1984.

[MKL+02] Dejan S. MILOJICIC, Vana KALOGERAKI, Rajan LUKOSE, Kiran NAGARAJA, Jim PRUYNE, Bruno RICHARD, Sami ROLLINS et Zhichen XU. « Peer-to-Peer Computing ». Technical Report HPL-2002-57, Hewlett-Packard, mars 2002. Disponible à `http://www.hpl.hp.com/techreports/2002/HPL-2002-57.html` (2003-08-12).

[MR01] Cyrille MARTIN et Olivier RICHARD. « Parallel Launcher for Cluster of PCs ». Dans *Proceedings of Parallel Computing (ParCo2001)*, septembre 2001. Disponible à `http://ka-tools.sf.net/` (2003-08-21).

[MS94] Steven A. MOYER et V. S. SUNDERAM. « PIOUS : A Scalable Parallel I/O System for Distributed Computing Environments ». Dans *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[MS02] David MAZIÈRES et Dennis SHASHA. « Building secure file systems out of Byzantine storage ». Dans *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, 2002.

[Mun01] Dan MUNTZ. « Building a Single Distributed File System from Many NFS Servers ». Technical Report HPL-2001-176, Hewlett-Packard, juillet 2001. Disponible à `http://www.hpl.hp.com/techreports/2001/HPL-2001-176.html` (2003-08-21).

[NK96] Nils NIEUWEJAAR et David KOTZ. « The Galley Parallel File System ». Dans *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, 1996. ACM Press.

[OD89] John K. OUSTERHOUT et Fred DOUGLIS. « Beating the I/O Bottleneck : A Case for Log-Structured File Systems ». *Operating Systems Review*, 23(1) :11–28, 1989.

[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 ». Dans *16th IEEE Mass Storage Systems Symposium held jointly with the 7th NASA Goddard Conference on Mass Storage Systems & Technologies*, San Diego, California, mars 1999. Parallel Computer Systems Laboratory, Dept. of ECE, University of Minnesota.

[PBB+00] K. W. PRESLAN, A. BARRY, J. BRASSOW, M. DECLERK, A.J. LEWIS, A. MANTHEI, B. MARZINSKI, E. NYGAARD, S. Van OORT, D. TEIGLAND, M. TILSTRA, S. WHITEHOUSE et M. O'KEEFE. « Scalability and Failure Recovery in a Linux Cluster File System ». Dans *Proceedings of the 4th Annual Linux Showcase and Conference*, College Park, Maryland, octobre 2000.

[PM96] Nadine PEYROUZE et Gilles MULLER. « FT-NFS : An Efficient Fault-Tolerant NFS Server Designed for Off-the-Shelf Workstations ». Dans *Symposium on Fault-Tolerant Computing*, pages 64–73, 1996.

[PR85] J. POSTEL et J. REYNOLDS. « File Transfer Protocol ». RFC765, octobre 1985.

[Pro00] The Globus PROJECT. « GridFTP : Universal Data Transfer for the Grid (White Paper) », septembre 2000.

[Pro02] The Globus PROJECT. « GridFTP Update », janvier 2002.

[PSB+01] Brian PAWLOWSKI, Spencer SHEPLER, Carl BEAME, Brent CALLAGHAN, Michael EISLER, David NOVECK, David ROBINSON et Robert THURLOW. « The NFS Version 4 Protocol ». Rapport Technique TR 3085, NetApp, 2000-2001. Disponible à `http://www.netapp.com/tech_library/3085.html`.

[RAM+01] Bruno RICHARD, Philippe AUGERAT, Nicolas MAILLARD, Simon DERR, Stéphane MARTIN et Céline ROBERT. « I-Cluster : Reaching TOP500 Performance Using Mainstream Hardware ». Technical Report HPL-2001-206, Hewlett-Packard, août 2001. Disponible à `http://www.hpl.hp.com/techreports/2001/HPL-2001-206.html` (2003-08-20).

[RCLL02] Robert B. ROSS, Philip H. CARNS, Walter B. LIGON III et Robert LATHAM. « Using the Parallel Virtual File System - User Guide », 2002. Disponible à `http://www.parl.clemson.edu/pvfs/user-guide.html`.

[RCN03] Bruno RICHARD, Denis CHALON et Donal Mac NIOCLAIS. « Clique : A transparent, peer-to-peer replicated file system ». Dans *Proceedings of the 4th International Conference on Mobile Data Management*, janvier 2003.

[Ric03] Bruno RICHARD. « *I-Cluster : Agrégation des ressources inexploitées d'un intranet et exploitation pour l'instanciation de services de calcul intensif* ». PhD thesis, Institut National Polytechnique de Grenoble, décembre 2003.

[RO92] Mendel ROSENBLUM et John K. OUSTERHOUT. « The Design and Implementation of a Log-Structured File System ». *ACM Transactions on Computer Systems*, 10(1) :26–52, 1992.

[Rob] Alan ROBERTSON. « High-Availability Linux Project ». Site web à `http://linux-ha.org/`.

[RT78] D. M. RITCHIE et K. THOMPSON. « The UNIX Time-Sharing System ». *The Bell System Technical Journal*, 57(6 (part 2)) :1905+, 1978.

[SDM98] Dominique SUEUR, Jean-Luc DEKEYSER et Philippe MARQUET. « DPFS : A Data-Parallel File System Environment ». Dans *HPCN Europe*, pages 940–942, 1998.

[SEP+97] S. SOLTIS, G. ERICKSON, K. PRESLAN, M. O'KEEFE et T. RUWART. « The Design and Performance of a Shared Disk File System for IRIX ». Dans *Proceedings of the Joint IEEE and NASA Mass Storage Conference*, 1997.

[SH02] Frank SCHMUCK et Roger HASKIN. « GPFS : A Shared-Disk File System for Large Computing Clusters ». Dans *Proceedings of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, janvier 2002. Disponible à `http://www.almaden.ibm.com/cs/gpfs.html` (2003-08-14).

[SHHP00] Oliver SPATSCHECK, Jørgen S. HANSEN, John H. HARTMAN et Larry L. PETERSON. « Optimizing TCP forwarder performance ». *IEEE/ACM Transactions on Networking*, 8(2) :146–157, 2000.

[Sil] SILICON GRAPHICS, INC.. « SGI CXFS : A High-Performance Multi-OS SAN Filesystem from SGI (White Paper) ». Disponible à `http://www.sgi.com/products/storage/cxfs.html` (2003-07-22).

[SK02] Yasushi SAITO et Christos KARAMANOLIS. « Name Space Consistency in the Pangaea Wide-Area File System ». Technical Memo HPL SSP Technical Memo No. 2002-12, Hewlett Packard, décembre 2002.

[SKKM02] Yasushi SAITO, Christos KARAMANOLIS, Magnus KARLSSON et Mallik MAHALINGAM. « Taming Aggressive Replication in the Pangaea Wide-Area File System ». Dans *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, décembre 2002.

[SMLN+02] Ion STOICA, Robert MORRIS, David LIBEN-NOWELL, David R. KARGER, M. Frans KAASHOEK, Frank DABEK et Hari BALAKRISHNAN. « Chord : A Scalable Peer-To-Peer Lookup Protocol for Internet Applications ». Dans *IEEE/ACM Transactions on Networking*, 2002.

[SPCSLCS00] Sistina SOFTWARE, University of Minnesota PARALLEL COMPUTER SYSTEMS LABORATORY, Dept. of ECE, Brocad COMMUNICATIONS et VERITAS SOFTWARE. « Implementing Journaling in a Linux Shared Disk File System ». Dans *Proceedings of the 8th NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with the 17th IEEE Symposium on Mass Storage Systems*, 2000.

[Sri95a] R. SRINIVASAN. « RPC : Remote Procedure Call Protocol Specification Version 2 ». RFC1831, août 1995.

[Sri95b] R. SRINIVASAN. « XDR : External Data Representation Standard ». RFC1832, août 1995.

[SRO96] Steven R. SOLTIS, Thomas M. RUWART et Matthew T. O'KEEFE. « The Global File System ». Dans *5th NASA Goddard Space Flight Center Conference on Mass Storage and Technologies*, College Park, Maryland, septembre 1996.

[SSB+95] T. STERLING, D. SAVARESE, D. J. BECKER, J. E. DORBAND, U. A. RANAWAKE et C. V. PACKER. « BEOWULF : A Parallel Workstation for Scientific Computation ». Dans *Proceedings of the 24th International Conference on Parallel Processing*, pages I :11–14, Oconomowoc, WI, 1995.

[Sun87] SUN MICROSYSTEMS, INC.. « XDR : External Data Representation Standard ». RFC1014, juin 1987.

[Sun88] SUN MICROSYSTEMS, INC.. « RPC : Remote Procedure Call Protocol Specification Version 2 ». RFC1057, juin 1988.

[Sun89] SUN MICROSYSTEMS, INC.. « NFS : Network File System Specification ». RFC1094, mars 1989.

[Tan03] Andrew TANENBAUM. *Systèmes d'exploitation - 2ème édition*. Pearson Education, 2003.

[TML97] Chandramohan A. THEKKATH, Timothy MANN et Edward K. LEE. « Frangipani : A Scalable Distributed File System ». Dans *Proceedings of the 16th ACM Symposium on Operating Systems*, octobre 1997.

[Tra01] Simon TRAVAGLIA. *The Bastard Operator from Hell - Version 1.0*. Plan Nine Publishing, mars 2001. Disponible à `http://bofh.ntk.net/Bastard.html` (2003-08-21)).

[Val02] Olivier VALENTIN. « Porting NFSP into Linux Kernel ». Rapport de stage de 2ème année, ENSIMAG, octobre 2002.

[Val03] Olivier VALENTIN. « Système de fichiers pour grilles de calcul ». Rapport de DEA, IMAG, juin 2003.

[vH00] Dimitri van HEESCH. « *GridFTP : FTP Extensions for the Grid* », 1997-2000.

[WA93] Randolph Y. WANG et Thomas E. ANDERSON. « xFS : A Wide Area Mass Storage File System ». Dans *Proceedings of the 4th Workshop on Workstation Operating System*, 1993.

[Wan03] An-I Andy WANG. « *The Conquest File System : A Disk/Persistent-RAM Hybrid Design for Better Performance and Simpler Data Paths* ». PhD thesis, University of California, Los Angeles, 2003. Disponible à `http://lasr.cs.ucla.edu/awang/papers/dissertation2003a.html` (2003-07-07).

[Wol98] Richard WOLSKI. « Dynamically forecasting network performance using the Network Weather Service ». *Cluster Computing*, 1(1) :119–132, 1998. Disponible à `http://www.cs.ucsd.edu/users/rich/publications.html` (2003-08-13).

[ZJQ+03] Yifeng ZHU, Hong JIANG, Xiao QIN, Dan FENG et David R. SWANSON. « Improved Read Performance in a Cost-Effective, Fault-Tolerant Parallel Virtual File System (CEFT-PVFS) ». Dans *Proceedings of IEEE/ACM CCGRID Workshop on Parallel I/O in Cluster Computing and Computational Grids*, 2003.

[ÁLL+] Rafael B. ÁVILA, Pierre LOMBARD, Adrien LEBRE, Yves DENNEULIN et Philippe O. A. NAVAUX. « Evaluation of Replication Model for a Distributed NFS Server ». Submitted to Cluster2003.

Annexe A

Glossaire

Définitions de termes

Nous avons au long de ce rapport essayé d'utiliser les termes français consacrés mais parfois la traduction disponible nous a semblé quelque peu obscure ou bien éloignée de la définition du terme aussi avons-nous préféré conserver le terme anglais.

snapshot Usuellement une image.
Une image de l'état d'un système à un instant donné qui permet éventuellement (si tout se passe bien) d'y revenir ultérieurement si une panne se produit.

distribution par bloc Voir figure A.1.
Si l'on considère une vue linéaire des données (taille S octets) et N machines, une distribution par bloc va placer sur chaque nœud $\frac{S}{N}$ octets. Le premier nœud contiendra les données de manière linéaire de 0 à $\frac{S}{N} - 1$, le second de $\frac{S}{N}$ à $2 \cdot \frac{S}{N}$-1, etc.

distribution cyclique Voir figure A.1.
Si l'on considère une vue linéaire des données (taille S octets) et N machines, une distribution cyclique va placer sur chaque nœud $\frac{S}{N}$ octets. Cependant, les données placées sur une machine ne vont pas être linéaires : la première machine (numéro 0) va stocker les données des *offsets* vérifiant $offset \bmod N = 0$, la seconde (numéro 1) $offset \bmod N = 1$, etc.

distribution par blocs cyclique Voir figure A.1.
Si l'on considère une vue linéaire des données (taille S octets) et N machines, une distribution par blocs cyclique va combiner comme son nom l'indique les deux distributions précédentes. Si la taille d'un bloc fait B octets, alors encore une fois chaque machine va contenir au total $\frac{S}{N}$ octets. La première machine (machine numéro 0) va contenir le bloc 0, le N, le $2 \cdot N$, ... La seconde machine contiendra les blocs 1, $N+1$, $2 \cdot N+1$, ... De manière générale, la machine j (entre 0 et $N-1$) contient les données situées entre $i \cdot N \cdot B + j \cdot B$ et $i \cdot N \cdot B + j \cdot B + B - 1$.

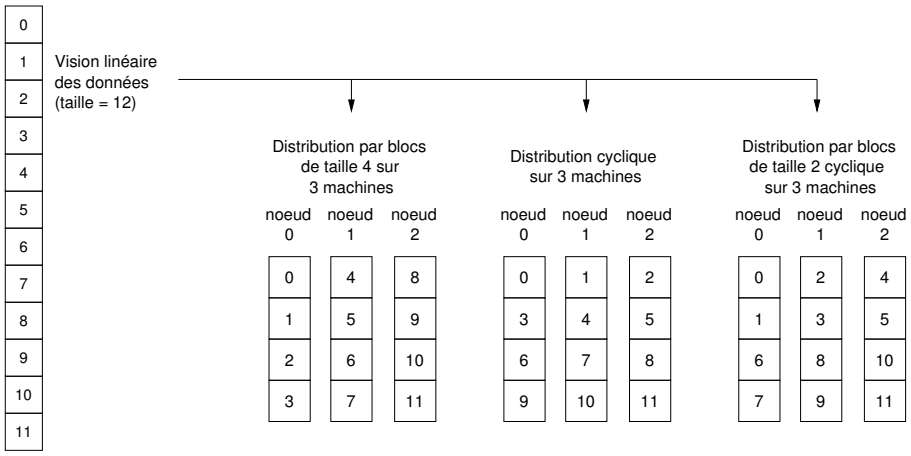

FIG. A.1 – Distribution des données

Dans le cas où $B = 1$, la distribution est alors cyclique. Il est à noter que pour le stockage, une distribution cyclique n'a que peu d'intérêts dans la mesure où le stockage se fait plus naturellement par blocs (par exemple sur un périphérique par blocs [*block device*]).

stripe Littéralement une bande (sous entendu : de données).

Dans un système de distribution cyclique, éventuellement par blocs, cette quantité désigne une bande de données telle que si une donnée de plus doit être écrite, elle le sera sur le premier nœud : le stripe i décrit donc les données entre $i \cdot N \cdot B$ et $i \cdot N \cdot B + N \cdot B - 1$.

Par exemple sur la figure A.1, sur le schéma du milieu (distribution cyclique) les données 0 à 2, 3 à 5, 6 à 8 et 9 à 11 constituent 4 *stripes*. Sur l'illustration de droite (distribution par blocs cyclique), les données 0 à 5 constituent le premier *stripe* et celles de 6 à 11 constituent le second. En reprenant la nomenclature des distributions cycliques précédentes, la constante $N \cdot B$ est appelée taille du *stripe*.

stride Littéralement : un pas.

Par exemple lors d'accès à des données, un motif d'accès à ces dernières peut être un accès *stridé*, c'est-à-dire que l'application doit accéder au contenu des fichiers va le faire en suivant un motif linéaire, c'est-à-dire que les accès vont se produire à des *offsets* respectant une fonction linéaire du type *offset0 + i · taille_du_pas*.

sparse file Un fichier creux.

Par analogie avec une matrice creuse qui contient beaucoup de zéros et peut donc utiliser des structures de données optimisées, ce type de fichier contient des « trous », c'est-à-dire que toutes le fichier n'a pas réellement son espace alloué sur le disque. Un tel fichier peut être obtenu par la séquence de pseudo commandes suivantes : création, déplacement à un *offset* non encore alloué puis écriture à partir de cet *offset*. Selon l'implantation du système, celui-ci peut tout allouer et mettre des zéros jusqu'à l'*offset* puis les données elles-mêmes ou bien ne conserver que les données à partir de l'*offset* allouées. Dans ce dernier cas, le

fichier est alors dit « creux » et lorsqu'une lecture sera demandée, sur une zone non encore écrite explicitement, le système devra remplacer la lecture des données non allouées par des zéros.

Notations

B Ce symbole désigne un octet (*byte*) soit 8 bits.
Les débits réseau étant souvent exprimés en bits/seconde alors que ceux des ma-té-riels (disques durs, par exemple) sont souvent exprimés en *byte*/seconde. Nous utiliserons en général dans ce document des débits en B/s (ou un de leur sur-multiples).

b Ce symbole désigne un bit.

E/S Entrées/Sorties (disque, réseau, ...)

I/O *Input/Output* = les Entrées/Sorties

Annexe B

Architecture de l'*i-cluster*

Le nom « i-cluster » désigne un projet mené par le laboratoire ID-IMAG et Hewlett Packard à partir de fin 2000. La fin prévue du projet devrait avoir lieu courant juin 2003, avec l'arrivée d'une nouvelle grappe d'Itanium-II sur le site de l'INRIA Montbonnot.

Un des axes de recherche et de développement de ce projet a été le développement d'outils de travail sur les grappes de nombreuses machines : l'environnement d'exécution parallèle (*Athapascan* et ses applications), les outils d'administration (*ka-tools*), les systèmes de fichiers (*NFSP*).

Un autres des buts de ce projet est d'étudier le fonctionnement de grappes virtuelles : ces grappes peuvent être des grappes temporaires que l'on peut trouver au sein d'une entreprise lorsque les machines sont inutilisées (la nuit par exemple). La gestion et l'utilisation de ces jachères de calcul posent un nombre de problèmes intéressants (détection des machines libres, sauvegarde et restauration du contexte de l'utilisateur, utilisation de techniques de « bacs à sables » [*sandboxing*]).

L'architecture du i-cluster a aussi permis d'avoir une modélisation simplifiée d'un intranet d'entreprise et de « voir » ce qu'il était possible de faire dans un tel environnement. Cette modélisation a été rendue possible par le fait que les machines ici rassemblées étaient des machines de bureau tout à fait dans la norme selon les critères de cette année 2000.

Les machines qui compose cette grappe sont des HP *i-vectra* équipés d'un processeur Intel Pentium III à 733MHz, de 256MB de RAM, d'un disque dur Maxtor IDE de 15GB et d'une carte réseau 3Com 100Mbit/s intégrée à la carte mère de type 3c905C.

Toutes les machines (d'abord 100 puis 225 début 2001) sont reliées entre elles par des *switches* HP Procurve 4000m. Chacun de ces périphériques dispose de 10 slots qui peuvent chacun avoir ou 1 port Gigabit ou 8 port Ether100.

Actuellement c'est un anneau simple gigabit comme illustré sur la figure B.1 page suivante. Cependant différents modèles de connexions et de routage entre les commutateurs `icluster-cX` ont été testés par le passé (maillage complètement connecté en pentacle, anneau double-gigabit, ... avec ou sans mode de routage *inter-switches*).

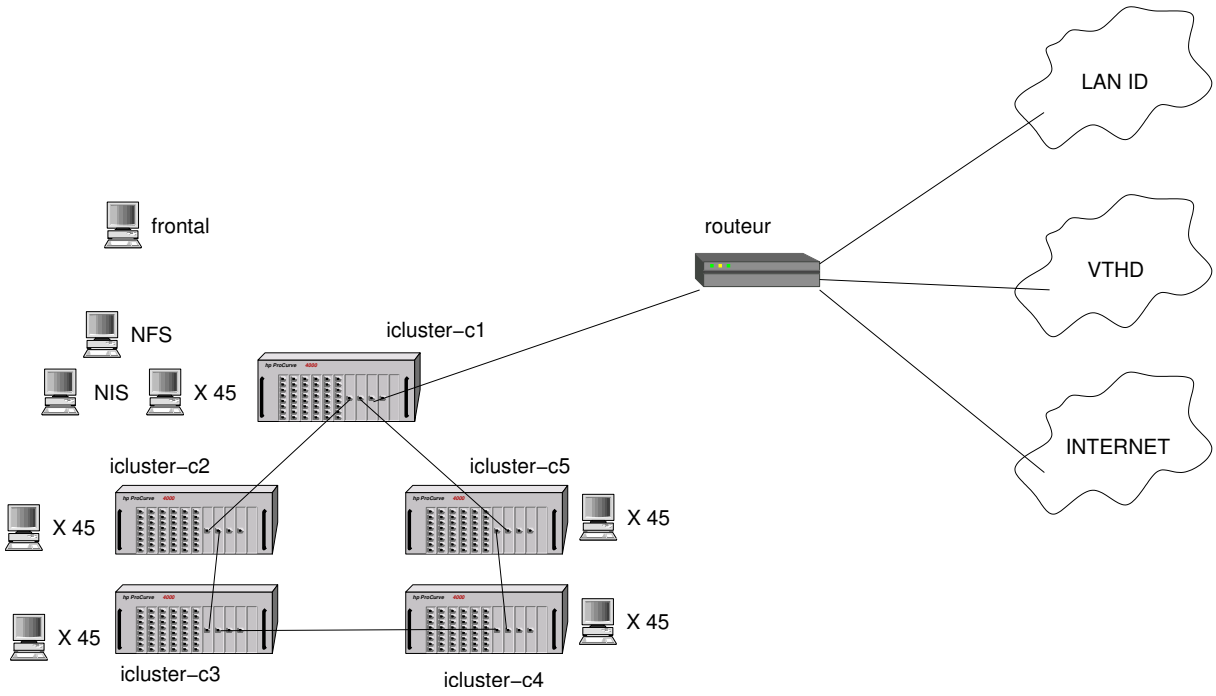

FIG. B.1 – Architecture de l'*i-cluster*

La grappe étant, de plus, reliée au réseau VTHD à 1Gb/s (ce qui prend actuellement un port gigabit sur l'un des commutateurs) une structure en simple anneau giga-bit a dû être retenue. Ainsi, chaque *switch* est relié à ses voisins par deux liens gigabit.

D'autre part en plus de son utilisation par les utilisateurs locaux, cette infrastructure a été ajoutée comme un membre actif du projet E-Toile de l'INRIA. Ce projet de 2 ans se déroulant entre début 2002 et fin 2003 vise à développer et à fournir une infrastructure d'intergiciels permettant d'expérimenter et d'évaluer diverses solutions techniques relatives aux grilles et grappes de grappes. De tels environnements qui tendent à émerger de plus en plus fréquemment ont en effet des problématiques, des contraintes et des spécifications qui ne sont pas les mêmes que celles observées sur un réseau plus classique tel que l'Internet et qui donc, requiert de nouvelles études.

Annexe C

Disponibilité des prototypes

Une (vieille) version du prototype est disponible sur le CD édité par l'INRIA en 2002. Son contenu est disponible à l'URL :

`http://pauillac.inria.fr/cdrom/prog/unix/fra.htm`

Les dernières versions des prototypes ainsi qu'une liste des diverses présentations et publications à propos de NFSP ou bien des outils développés au dessus de cette brique logicielle se trouvent à l'URL :

`http://www-id.imag.fr/Laboratoire/Membres/Lombard_Pierre/nfsp/`

Le prototype de transferts parallèles, GXfer, devrait être bientôt disponible par le biais du site du projet RNTL E-Toile :

`http://www.urec.cnrs.fr/etoile/`

Résumé

Le stockage de données utilise souvent des systèmes se caractérisant par une grande intrusivité : ceux-ci requièrent de nombreuses modifications logicielles, voire parfois même matérielles, pour être déployés et utilisés. Notre solution consiste à offrir un stockage distribué logiciel pour architectures de type grappes de nature faiblement intrusive dans la mesure où le protocole standard omni-présent du monde Unix, NFS, est utilisé. L'approche retenue se caractérise par une séparation de la gestion des méta-données et des données permettant ainsi de répartir la charge d'entrées/sorties et d'obtenir de meilleures performances. L'ajout de redondance permet aussi de disposer à moindre coût de stockage distribué encore plus performant et plus sûr. Le développement d'outils de transfert efficace inter-grappes et d'un système distribué de fichiers à plus grande échelle a permis de valider notre approche.

Mots clés : NFS - stockage - grappe - systèmes de fichiers - distribution

Abstract

Data storage often uses systems characterized by an important intrusivity : these require numerous software alterations, even sometimes hardware, to be deployed and used. Our solution consists in offering a distributed storage (thanks to software) for clusters with a very light intrusivity as the standard ubiquitous protocol of the Unix world, NFS, is used. Our approach uses a separation in the management of meta-data and data to load balance the I/O load and obtain better performances. The redundancy added lets the user have at a small cost a distributed storage even more performant and reliable. Efficient inter-clusters transfer tools and the use of our solution in even bigger-scale distributed file system have allowed us to validate the design choices.

Keywords : NFS - storage - cluster - file system - distribution