Un canevas logiciel pour la construction de systèmes de stockage reconfigurables pour grappes de machines

Renaud Lachaize

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 LSR-IMAG, projet SARDES,
dans le cadre de l'Ecole Doctorale
« Mathématiques Sciences et Technologies de l'Information »

présentée et soutenue publiquement par

Renaud LACHAIZE

le 14 Septembre 2005

Un canevas logiciel pour la construction de systèmes de stockage reconfigurables pour grappes de machines

---

Directeur de thèse :

Jacques MOSSIÈRE

JURY

M.	Roger	MOHR	Président
Mme.	Christine	MORIN	Rapporteur
M.	Raymond	NAMYST	Rapporteur
M.	Yves	DENNEULIN	Examinateur
M.	Jørgen Sværke	HANSEN	Examinateur
M.	Jacques	MOSSIÈRE	Directeur de thèse

À mes parents
À mes grands-parents

Résumé

Cette thèse s'intéresse aux systèmes de stockage répartis pour grappes de serveurs. Les solutions existantes sont généralement monolithiques et peu (re)configurables. Elles limitent la réutilisation de code, compliquent l'administration et le développement de systèmes robustes et autonomes. Nous proposons un canevas logiciel visant à lever ces contraintes. Un service de stockage est construit à partir d'un assemblage de composants et fournit une représentation explicite des flux de contrôle et de données. L'architecture permet un paramétrage fin des propriétés du système, y compris au niveau des protocoles de communication et des stratégies de transfert des données. L'infrastructure d'exécution offre, de plus, des mécanismes de reconfiguration dynamique du code, des paramètres et de la structure du système. Un prototype, implémenté au niveau bloc, montre qu'une approche modulaire est conciliable avec de bonnes performances et permet de construire des services flexibles et spécialisés.

Abstract

This thesis focuses on distributed storage systems for clusters of servers. Existing solutions are generally monolithic and hardly (re)configurable. As a consequence, code reuse is limited, administration of such services is complex, and developing robust and autonomic systems is difficult. We propose a software framework to alleviate these constraints. A storage service is constructed from a set of components and provides an explicit representation of control and data streams. The architecture allows a fine-grained tuning of the system properties, including networking protocols and strategies for data transfers. In addition, the execution infrastructure provides mechanisms for dynamic reconfiguration of the code, the parameters and the structure of the system. A prototype was implemented at the block level. It shows that a modular approach can help building flexible and custom systems and is compatible with good performance.

Remerciements

Mes premiers remerciements sont adressés à Roger Mohr, Professeur à l'Institut National Polytechnique de Grenoble, pour m'avoir fait l'honneur de présider ce jury.

Je remercie Christine Morin, Directrice de recherches à l'Inria et Raymond Namyst, Professeur à l'Université de Bordeaux 1, d'avoir accepté le rôle de rapporteur de cette thèse et évalué mon travail de manière approfondie et constructive. Merci également à Yves Denneulin, Maître de Conférences à l'Institut National Polytechnique de Grenoble pour sa participation à ce jury et les échanges que nous avons eus au cours des dernières années.

Je souhaite bien sûr remercier chaleureusement Jørgen Sværke Hansen, Associate Professor à l'Université de Copenhague, à l'initiative du projet Proboscis, pour m'avoir pris sous son aile et accompagné tout au long de cette thèse. Les idées et les résultats exposés dans ce document lui doivent beaucoup.

J'exprime ma gratitude à Jacques Mossière, Professeur à l'Institut National Polytechnique de Grenoble, mon directeur de thèse, pour son encadrement, ses nombreux conseils, sa disponibilité, son soutien et sa patience à mon égard. Son influence sur mon parcours universitaire a été décisive : il fut à l'origine de ma découverte de l'algorithmique, des systèmes d'exploitation, et de l'activité d'enseignant chercheur.

Au delà du jury, je tiens à remercier tous ceux qui ont permis à ces travaux d'aboutir, par leurs conseils, leurs contributions et leurs encouragements.

- Merci à tous les enseignants chercheurs qui ont su, par leurs qualités pédagogiques et techniques, attiser mon intérêt pour l'informatique et les systèmes répartis. J'exprime, en particulier, mon profond respect à Sacha Krakowiak et à Xavier Rousset de Pina.
- Merci à ce dernier et à Luc Bellissard de m'avoir aidé à m'orienter vers un DEA et une thèse.
- Merci à Roland Balter et à Jean-Bernard Stefani de m'avoir accueilli au sein des projets Sirac et Sardes.
- Merci à Emmanuel Cecchet pour ses nombreux coups de mains, sa bienveillance et son énergie communicative.
- Merci à Sébastien Jean pour ses conseils, son indéfectible bonne humeur et son aide précieuse en des moments clés.
- Merci à Christophe, Simon et Aurélien, qui furent mes principaux compagnons lors de mes deux premières années de thèse, pour tout ce qu'ils m'ont appris, leur aide et les bons moments partagés.

- Merci à Takoua et Oussama, mes camarades de bureau, d'avoir supporté mes humeurs variables et mes petites manies.
- Merci à Valérie Gardès, Catherine Magnin et Élodie Toihein, assistantes du projet Sardes, pour leur gentillesse et la redoutable efficacité avec laquelle elles sont toujours parvenues à démêler de nombreuses tracasseries administratives.
- Merci à toutes les personnes sympathiques et intéressantes que j'ai eu l'occasion de côtoyer aux cours de ces dernières années (au sein de Sardes, de l'Inria, des laboratoires LSR et ID, de l'Ensimag, de l'IUT de Valence ou ailleurs) et qui voudront bien m'excuser de ne pas les citer nommément. Je commence doucement à prendre conscience de la chance insolente qui m'a permis d'évoluer dans des conditions matérielles et un environnement humain plus qu'agréables.

Enfin, je garde un place toute particulière pour ma famille et mes proches, pour tout ce qu'ils ont pu (et continuent à) m'apporter, en dépit de mon caractère difficile, et notamment à Cécile, pour sa présence inestimable à mes côtés.

Table des matières

Table des figures

Introduction

Motivation et objectifs

Le recours massif à la gestion de données numérisées, principalement à des fins de manipulation automatisée et d'archivage, ne se dément pas. Cette tendance est à la source de nombreux défis technologiques à différents niveaux : acquisition, acheminement, analyse/traitement et stockage. Ce dernier aspect est crucial pour au moins deux raisons majeures. D'une part, pour son rôle intrinsèque : assurer la permanence des informations pour un usage futur. D'autre part, car les technologies actuelles de stockage, limitées par des paramètres mécaniques, constituent l'un des principaux goulots d'étranglement d'un système informatique.

Pour faire face aux volumes de données à traiter, les architectures monolithiques ont progressivement été abandonnées au profit d'infrastructures réparties interconnectant un grand nombre de serveurs coopératifs. La première étape de cette transition majeure s'est concrétisée par une mise en œuvre dans un environnement local, fortement couplé et a abouti au concept de *grappes de machines*. Des efforts significatifs, regroupés sous l'étiquette de *Grid computing*[1], visent maintenant à agréger des ressources dans un cadre plus global, à la fois en termes de répartition géographique (échelle d'un pays, voire de la planète) et d'hétérogénéité des plates-formes (de l'ordinateur de poche à un ensemble de grappes).

Si le chantier des « grilles de calcul » est déjà bien avancé, cela ne signifie pas pour autant que le contexte plus restreint des grappes est parfaitement maîtrisé et n'offre plus de défis, bien au contraire. L'augmentation régulière du nombre de nœuds[2] (avec pour buts une vitesse de calcul/traitement accrue, une meilleure résistance aux pics de charge et davantage de tolérance aux pannes) accentue les difficultés de développement et d'optimisation des applications réparties. Il en va de même pour l'administration globale du système permettant de détecter d'éventuelles pannes ou problèmes de performance et d'y réagir. En conséquence, les coûts humains nécessaires au bon fonctionnement d'une grappe deviennent préoccupants. Pire, de nombreuses études industrielles établissent qu'une majorité des problèmes critiques sont en fait imputables à des erreurs d'administrateurs humains, bénéficiant pourtant d'une formation spécialisée. Il devient donc nécessaire d'envisager la construction de systèmes capables, au moins partiellement, d'auto-administration.

Cette thèse s'intéresse au thème des systèmes logiciels de stockage pour grappes de machines. Ce sujet a été abondamment étudié au cours des quinze dernières années, en particulier sous trois angles principaux : augmentation de la robustesse par rapport aux pertes de données, amélioration des performances d'accès aux données et enfin, conséquence des deux axes précédents, gestion répartie du service de stockage.

[1] L'expression découle d'une analogie avec le réseau de production électrique, pris comme modèle de simplicité (du point de vue des utilisateurs), d'ubiquité et de montée en charge.

[2] L'ordre de grandeur du millier de nœuds est désormais courant.

En revanche, beaucoup moins de travaux ont cherché à faciliter la construction de systèmes de stockage *flexibles*, notion que nous tentons d'expliciter par les trois dimensions ci-dessous :

Configurabilité : possibilité de configurer finement les paramètres du système pour optimiser son fonctionnement dans un contexte opérationnel particulier ;

Modularité : assemblage du système de stockage à partir d'une bibliothèque de modules logiciels spécialisés afin de simplifier la mise en œuvre d'un système de stockage spécialisé ;

Reconfiguration dynamique : possibilité d'influer sur le comportement et la composition du système en cours de fonctionnement.

Ces aspects sont pourtant cruciaux car (1) les besoins des utilisateurs sont très variés, (2) la mise au point d'un service de stockage de bas niveau, dont la fiabilité et les performances sont critiques, est généralement fastidieuse, et (3) les capacités de reconfiguration dynamique constituent un pré-requis important pour bâtir des systèmes hautement disponibles.

L'objectif des travaux présentés dans cette thèse est de contribuer à l'émergence de systèmes de stockage flexibles, qui nous semblent constituer une étape nécessaire vers la réalisation de systèmes auto-administrables. Nos recherches se sont focalisées sur les couches basses (échanges de données entre machines et périphériques de stockage) afin de démontrer que l'introduction de souplesse aux sein de fonctions critiques n'est pas incompatible avec des critères élevés de performance.

De manière plus précise, nous nous sommes intéressés à trois axes principaux :

- la définition d'**un modèle à composants spécialisé** pour les systèmes de stockage réparti, afin de mieux structurer les interactions entre les différents modules fonctionnels qui composent le système, de permettre la réutilisation de code et de simplifier le déploiement de services optimisés au sein d'une grappe ;
- l'élaboration d'**un modèle de programmation pour l'échange de données entre nœuds** permettant de configurer le protocole et la stratégie de communication de façon dynamique, sans modifier le code du service de stockage, pour accroître l'adaptabilité du système ;
- le développement de **techniques de reconfiguration dynamique** autorisant des modifications profondes du système en cours de fonctionnement (code, paramètres, structure) et de capacités élémentaires d'introspection.

Cadre du travail

Cette thèse a été effectuée dans le cadre du projet Sardes (*System Architecture for Reflexive Distributed EnvironmentS*) dont le principal objectif est l'étude d'architectures et de méthodes de construction d'environnements informatiques répartis au sens large (grande diversité de ressources, d'échelles et de mécanismes de communication). L'approche adoptée pour l'élaboration de telles infrastructures repose sur l'emploi systématique de techniques de programmation par composants et de réflexivité (capacité d'un système à s'introspecter et à opérer sur lui-même).

Les recherches sur les systèmes de stockage flexibles pour grappes ont débuté dans le contexte du séjour post-doctoral de Jørgen Hansen au sein de Sardes. Le travail présenté dans ce manuscrit a largement bénéficié des jalons tant bibliographiques que pratiques déjà posés par ce dernier avant notre arrivée, ainsi que de son suivi tout au long de cette thèse.

Démarche suivie

Nous avons commencé par poursuivre les pistes existantes sur l'emploi de ressources banalisées pour la construction de systèmes de stockage efficaces en travaillant sur deux fronts principaux. Nous avons, tout d'abord, étudié la viabilité d'une approche où chaque nœud agit comme entité de calcul/-traitement « classique » tout en mutualisant une partie des ses ressources pour participer à la mise en œuvre d'un service de stockage global. Par ailleurs, nous avons cherché à définir une approche plus structurée et systématique pour la construction et le déploiement d'un tel service. Une première évaluation dans un contexte réaliste (réseau de stockage « virtuel » déployé en dessous d'un système de fichiers partagé) nous a permis de conclure à la compatibilité de notre approche avec des exigences fortes en termes de performances [90].

Par la suite, nous avons cherché à étendre la flexibilité de notre modèle au niveau des stratégies de transfert de données employées [129] et ajouté le support d'une nouvelle interface de communication (TCP/IP) à notre prototype qui ne supportait au départ qu'une interface de bas niveau pour réseaux SCI. Dans une troisième phase, nos efforts ont porté sur l'introduction de mécanismes de reconfiguration dynamique [128]. Nous avons enfin développé différents services de stockage réparti pour compléter notre évaluation.

Plan du document

Ce manuscrit est organisé en sept chapitres. Les trois premiers correspondent à l'état de l'art ; un lecteur familier avec les domaines abordés (et relativement en phase avec notre terminologie) devrait pouvoir ignorer les deux premiers sans grand préjudice pour la compréhension de la suite.

Le premier chapitre résume l'évolution des grappes de machines depuis une dizaine d'années. Il met à la fois l'accent sur les applications déployées sur ce type d'environnement mais également sur l'état actuel des technologies relatives aux principaux composants matériels d'une telle plate-forme : les microprocesseurs et les interfaces de communication.

Le second chapitre est axé sur les techniques de stockage de données, principalement au niveau logiciel. Il commence par introduire les notions majeures dans le contexte d'un système centralisé. Dans un second temps, il décrit les solutions développées dans le contexte des grappes de serveurs, en fonction des différents niveaux d'interface considérés.

Le troisième chapitre achève notre examen de l'état de l'art en présentant les systèmes de stockage flexibles développés jusqu'à présent et en identifiant leurs lacunes. Il nous permet d'expliciter le positionnement ainsi que les principaux objectifs de notre proposition.

Le quatrième chapitre présente le modèle à composants que nous avons défini. Celui-ci permet de développer des systèmes de stockage répartis de bas niveau par assemblage de composants, selon l'abstraction de « chemin d'entrées/sorties ». Les modèles de composition, de programmation et d'exécution y sont notamment détaillés.

Le cinquième chapitre introduit un modèle de programmation (et l'infrastructure logicielle associée) visant à découpler l'implémentation d'un serveur réparti de données et les stratégies de transfert qui lui sont associées afin de pouvoir adapter une telle application à des contraintes aussi bien statiques (contexte de déploiement particulier) que dynamiques (évolution des conditions de charge).

Le sixième chapitre traite des mécanismes de reconfiguration dynamique rendus possibles par les choix de conception du canevas logiciel présenté précédemment. Il montre comment un système de stockage basé sur une structure à composants peut subir des modifications de code, de paramètres et de structure sans interruption de service vis à vis des couches clientes.

Le septième chapitre expose un résumé des principaux résultats expérimentaux obtenus avec notre prototype développé au niveau d'interface blocs.

Ce dernier chapitre est suivi d'une conclusion qui résume nos contributions et évoque les perspectives de recherche qu'elles ouvrent.

Chapitre 1

Grappes de machines et technologies d'interconnexion

Sommaire

1.1 Introduction

En une dizaine d'années, les grappes de machines se sont imposées comme une solution particulièrement attractive en termes de rapport puissance/coût et ont nettement modifié le panorama des centres de calcul. Ce bouleversement est la conséquence de deux facteurs antérieurs : la révolution de l'informatique personnelle et la généralisation des réseaux de communication.

L'arrivée de l'ordinateur personnel (PC) au début des années 1980 a créé un cercle vertueux qui ne se dément pas depuis vingt ans : les microprocesseurs disponibles sur le marché sont de plus en plus puissants, bénéficient d'une densité d'intégration sans cesse accrue[1], le prix d'une configuration donnée baisse jusqu'à 30% dans l'année qui suit sa sortie [227] et le prix d'un équipement haut de gamme diminue légèrement d'année en année. En conséquence, une machine est aujourd'hui considérée comme obsolète trois ans après sa sortie et la puissance d'un PC actuel dépasse celle d'un supercalculateur d'il y a dix ans.

Jusqu'au début des années 1990, les réseaux locaux d'interconnexion de machines n'ont pas bénéficié de progrès frénétiques dans les mêmes proportions, ni au même rythme que les microprocesseurs. Leur usage s'est cependant généralisé progressivement à partir du milieu des années 1980 (autour d'un standard, Ethernet), mouvement accéléré par le phénomène Internet au début de la décennie suivante.

[1] Doublement du nombre de transistors pour un circuit de même taille tous les 18 mois, selon la célèbre loi empirique de Gordon Moore, encore valable aujourd'hui.

Malgré l'écart important entre les performances des processeurs et celles des réseaux, l'utilisation de plusieurs machines interconnectées s'est avérée économiquement séduisante pour résoudre des problèmes de grande taille (trop importante pour un unique PC). C'est ainsi que sont apparus les *Networks of Workstations* (NOW) ou réseaux de stations de travail à la fin des années 1980. Il s'agissait d'utiliser une partie des ressources (cycles des processeurs et bande passante du réseau) d'un réseau d'entreprise pour paralléliser l'exécution de calculs. Les NOW, de par leur faible coût, ont petit à petit gagné en popularité même si les performances restaient bien en deçà de celles des supercalculateurs du moment. La diffusion d'environnements logiciels visant à simplifier la programmation et optimiser l'efficacité des NOW a accentué cet état de fait.

Au début des années 1990, certains laboratoires, tels que le CESDIS (*Center of Excellence in Space Data and Information Sciences*) [210] [209] ont étendu le principe des NOW à des machines entièrement consacrées à l'exécution d'une application distribuée, c'est à dire sans partage des ressources avec d'autres applications (interactives, par exemple). C'est ce type d'emploi de machines en réseau que désigne généralement le terme de grappe ou *cluster* [139].

Ce chapitre commence par présenter les composants et les schémas d'utilisation courants des grappes. Dans un second temps, il détaille les technologies d'interconnexion utilisées dans ce contexte.

1.2 Principales caractéristiques des grappes

Dans cette section, nous présentons les composants matériels d'une grappe ainsi que les principaux besoins des utilisateurs.

1.2.1 Composants matériels

Les machines d'une grappe (ou *nœuds*) sont généralement des stations de travail « haut de gamme », même si des réalisations basées sur des PC de bureau [186] se sont montrées viables et relativement efficaces. L'usage de telles configurations ne se justifie pas uniquement par la puissance des processeurs mais aussi pour pouvoir bénéficier des cartes mères optimisées. En effet, le processeur n'est que rarement l'élément limitant des performances. Les goulots d'étranglement se situent au niveau des périphériques (disques durs, interfaces réseau) et parfois à celui du bus mémoire. Ce constat, valable au niveau d'une machine, l'est encore d'avantage à l'échelle d'une grappe, où l'efficacité des communications réseau a généralement un impact dominant sur les performances ; le type et la topologie du réseau sont un critère déterminant de l'efficacité d'une grappe. Ainsi, l'utilisation de stations de travail permet de bénéficier d'interconnexions plus efficaces au niveau de la mémoire et des périphériques d'Entrées/Sorties (E/S). De manière analogue, l'emploi de processeurs 64 bits se justifie souvent autant par des besoins en matière de capacité d'adressage qu'en termes de performances du CPU et de la carte mère.

L'approche privilégiant avant tout l'efficacité est majoritaire aujourd'hui mais d'autres critères commencent à gagner en importance parallèlement à l'accroissement du nombre moyen de nœuds. D'une part, le retour sur investissement d'une grappe est rendu difficile par la rapide obsolescence du matériel (3 ans voire moins). Ceci incite les responsables de grandes grappes, comme le moteur de recherche Google, à s'équiper de machines « milieu de gamme » pour maximiser le rapport performance/prix [22]. D'autre part, les contraintes de volume et de puissance thermique ne sont plus négligeables et motivent (de manière intrinsèque et/ou par les coûts qu'elles induisent) des investigations autour d'architectures optimisées pour ces aspects [70].

La plupart des grappes actuelles sont composées de plusieurs centaines de nœuds mono ou biprocesseurs, équipés d'un ou plusieurs Go de mémoire, interconnectés par un réseau rapide (le plus

souvent Myrinet ou Gigabit Ethernet) pour les communications applicatives et par un réseau plus lent (Fast Ethernet, généralement) pour les tâches d'administration.

Evolution des processeurs Jusqu'à une période relativement récente, les stations de travail ont été basées selon un modèle mono ou bi-processeur. Les architectures offrant plus de parallélisme matériel (quatre, huit voire plusieurs dizaines de processeurs) ont, compte-tenu de leur prix, été réservées aux serveurs spécialisés (supercalculateurs « à la SGI », gestion de transactions, etc.). Ce constat est naturellement transposable aux nœuds qui composent une grappe[2].

Dans ce contexte, l'accroissement de la puissance de calcul au fil des générations de machines a été réalisée grâce à l'amélioration régulière de deux paramètres principaux : la capacité des caches et la fréquence de fonctionnement. Le premier paramètre est une conséquence directe de la loi de Moore et continue d'être un levier efficace (et compense en partie le fossé croissant entre la vitesse des processeurs et la latence des mémoires RAM). En revanche, l'accélération frénétique des horloges que nous avons connue appartient désormais au passé [215]. En effet, des problèmes jusqu'alors surmontables (dissipation thermique, consommation et fuites électriques) sont progressivement devenus épineux, étant données les fréquences et les dimensions imposées aux circuits actuels. Cette « cassure » est perceptible depuis 2002[3].

À l'heure actuelle, tous les fabricants de processeurs ont reconnu ce tournant historique et modifié leur stratégie. Plutôt que de chercher à augmenter à tout prix la fréquence d'exécution, les efforts sont désormais tournés principalement vers la gestion du parallélisme d'exécution au sein d'une même puce (*Chip Multi-Threading* — CMT). Le support du parallélisme interne repose sur deux méthodes complémentaires : la gestion simultanée de plusieurs contextes d'exécution au sein d'une unité d'exécution (Simultaneous Multi-Threading – SMT) [224] et l'intégration de plusieurs unités d'exécution au sein d'une même puce (*Chip MultiProcessing* — CMP) [208].

Un processeur SMT, dans la lignée des processeurs superscalaires, est capable d'exécuter plusieurs instructions par cycle et possède les jeux de registres nécessaires pour gérer simultanément plusieurs contextes. Ainsi, le parallélisme inter-threads est converti en du parallélisme d'instructions et permet une meilleure utilisation des ressources du CPU, sans induire de grosses contraintes de conception, de taille, de coût du processeur ni d'importantes modifications des systèmes d'exploitation. Des études ont montré que des applications parallèles pouvaient bénéficier d'accélérations significatives (gain de 5 à 30% en temps d'exécution) sur une architecture SMT. Les techniques SMT sont déjà assez largement déployées aujourd'hui, notamment au travers de la technologie Hyperthreading d'Intel [113], dans les processeurs Xeon et Pentium 4.

L'intérêt de l'approche CMP réside dans l'intégration au sein d'un même circuit de plusieurs unités d'exécution. Les mécanismes de communication et de synchronisation entre CPU sont ainsi optimisés. Il en résulte une meilleure utilisation des ressources (partage d'un ou plusieurs niveaux de cache, multiplexage efficace des accès à la mémoire, etc.). Le recours à cette seconde technique est plus récent mais cet axe d'optimisation est désormais la priorité des constructeurs. Les processeurs CMP actuels sont basés sur deux « cœurs » mais des modèles avec huit cœurs SMT sont annoncés dans un futur proche.

[2]Une exception notable concerne bien sûr les grappes de supercalculateurs, minoritaires mais assez bien placées au palmarès des ordinateurs les plus puissants. Dans un cas extrême, une telle topologie est nommée *constellation* et comporte un nombre de processeurs par nœud supérieur au nombre total de nœuds.

[3]En l'absence de ralentissement, les machines auraient atteint la barrière des 10 GHz en 2005. En Mars 2005, celle des 4 Ghz n'avait pas encore été franchie.

Ces évolutions récentes des processeurs ont un impact important. Même à nombre de nœuds (et de processeurs par nœud) constant, une grappe de stations va offrir, au fil des générations de circuits, de plus en plus de support matériel pour le parallélisme d'exécution. Ce constat incite, d'une part, à structurer les applications de manière toujours plus parallèle et à optimiser les ordonnanceurs des systèmes d'exploitation et des intergiciels à cet effet. D'autre part, cette perspective fournit également une opportunité de reconsidérer la façon dont un système d'exploitation se charge de ses missions, en particulier au niveau des communications et du stockage de données.

1.2.2 Profils d'utilisation

A l'instar des premiers calculateurs, les premières grappes furent surtout employées pour des applications de calcul et de simulations scientifiques. L'explosion récente du recours aux services en ligne et du volume des données associées ont favorisé par la suite l'émergence de serveurs de données en grappe. Nous présentons ici ces différents profils d'utilisation.

1.2.2.1 Applications scientifiques

Les applications scientifiques sont généralement intensives en calculs (manipulations de matrices, calculs flottants...) ainsi qu'au niveau de la quantité de données traitées [160]. Elles ne nécessitent pas d'interactivité et sont donc lancées en mode « batch », une expérience après l'autre. Leur critère premier est la vitesse d'exécution ; les besoins sont moindres en termes de tolérance aux pannes (les points de reprise sont souvent considérés comme une mesure suffisante) et de dynamisme (les ressources matérielles mobilisées sont fixées au lancement de l'application). Par conséquent, dans le but d'optimiser les performances, une attention particulière est portée au placement et à l'accès aux données (en mémoire et sur disque). L'émergence d'interfaces de programmation telles que MPI-IO [147] témoigne de ces efforts.

Le déploiement sur grappe d'une application de calcul est généralement symétrique : à l'exception d'éventuels nœuds de stockage et d'un coordinateur, tous les nœuds jouent le même rôle et résolvent une partie du calcul global en travaillant sur un sous-ensemble des données. Pour des raisons de performances, ce découpage s'effectue à relativement gros grain. Il est en effet plus difficile d'exploiter efficacement les ressources d'une grappe que celles d'un supercalculateur monolithique en raison, d'une part, des coûts beaucoup plus élevés des communications interprocesseurs, et d'autre part, de la gestion complexe de la distribution dans le code applicatif. De nombreux efforts ont donc cherché à fournir des environnements à « image unique » (SSI pour *Single System Image*) qui simplifier la tâche des programmeurs en gérant tout ou partie de ces contraintes et en fournissant l'abstraction d'une seule machine.

Les SSI (tels que OpenSSI, OpenMosix ou Kerrighed [136]) nécessitent généralement de profondes modifications du noyau du système d'exploitation sur lequel ils sont basés ainsi qu'un fort couplage entre les différents nœuds, afin de fournir des caractéristiques telles que la migration de processus et l'accès global à un espace de mémoire virtuelle partagé (*Distributed Shared Memory*). En contrepartie, un SSI peut exécuter une application parallèle (écrite à la base pour une machine multi-processeurs) sur une grappe sans aucune modification ni recompilation de code.

1.2.2.2 Serveurs de données

Les serveurs de données en grappe sont aujourd'hui utilisés pour des besoins variés : serveurs Web, bases de données [44], fouille de données (moteurs de recherche [22] et data mining), services multimedia tels que la vidéo à la demande [114], messagerie électronique [189], etc.

Les serveurs de données partagent avec les applications scientifiques l'exigence de performances mais en diffèrent par la prise en compte d'un autre critère majeur : la haute disponibilité. De nombreux services se doivent d'être continuellement disponibles ; d'importantes capacités de tolérance aux fautes sont donc nécessaires, et en cas d'incident, il est généralement préférable de continuer à fournir un service (avec une qualité et des performances éventuellement dégradées) plutôt que de l'interrompre. Ces contraintes induisent de forts besoins de reconfiguration dynamique.

Une autre différence avec le monde du calcul tient au fait que les serveurs de données en grappe sont généralement organisés en plusieurs étages fonctionnels hiérarchiques. Par exemple, certains serveurs de video à la demande [193] comprennent un ensemble de nœuds consacrés au stockage de masse, un second ensemble servant de cache de films et un dernier gérant la diffusion des images aux clients. Dans la même logique, l'infrastructure des sites de commerce électronique est basée sur des architectures [43] qui définissent plusieurs fonctions distinctes :

- **Serveur web :** la gestion des connexions web et le service des pages web statiques,
- **Serveur de présentation :** la génération de pages web dynamiques,
- **Serveur d'applications :** la logique applicative, le « cœur de métier » du site (gestion des commandes, etc.) ;
- **Base de données :** la persistance des données.

L'infrastructure des grands serveurs de données est basée sur l'utilisation de grappes pour chacune des couches fonctionnelles, à la fois pour des motivations de passage à l'échelle (traitement parallèle des requêtes) et de tolérance aux fautes (duplication des composants matériels et logiciels).

Dans le contexte des grappes de serveurs de données, les besoins en matière de « système à image unique » au niveau global sont moindres que pour les applications de calcul du fait du découpage explicite en plusieurs étages fonctionnels. Cependant, une préoccupation importante des administrateurs concerne la reconfiguration dynamique d'un étage donné (ajout et retrait de nœuds sans perturber le fonctionnement de l'application).

1.2.2.3 Usages mixtes et dynamiques

Les sections précédentes ont abordé deux usages typiques des grappes : calculateur parallèle et serveur de données. Ces deux profils d'utilisation bien définis, et relativement statiques, s'avèrent courants dans des contextes industriels mais de nombreuses grappes (pour la plupart expérimentales) ne peuvent être associées exclusivement à l'un ou l'autre de ces rôles et imposent des contraintes d'exploitation supplémentaires. C'est notamment le cas des plates-formes universitaires mises à disposition d'une grande communauté d'utilisateurs (par exemple NetBed [233], un banc d'essai à grande échelle pour les chercheurs en systèmes répartis et réseaux). C'est également le cas des centres d'hébergement (*« hosting centers »*) [48] qui fournissent l'infrastructure d'exécution (matérielle et logicielle) nécessaire à de nombreux sites web commerciaux aux besoins variés, en termes de charge supportée et de services offerts.

Dans de tels contextes, une grappe est partagée dynamiquement en sous ensembles, alloués à différents utilisateurs. La taille (en nombre de nœuds) et la durée des réservations peuvent être très variables (de quelques dizaines de minutes à plusieurs jours voire semaines). En outre, le spectre des applications déployées à un instant donné sur la grappe peut être très large : dans certains cas, la grappe n'est utilisée que comme un super-ordinateur partitionné en plusieurs sous-groupes de configurations homogènes (pour du calcul ou des services web), dans d'autres, la grappe est utilisée comme un serveur généraliste qui exécute des applications variées (calculs, expériences sur des bases de données, évaluation de performances de réseaux, débogage d'une application répartie...). Des utilisateurs

différents peuvent avoir besoin de configurations sensiblement différentes, en termes de logiciels, d'intergiciels, de système d'exploitation, voire de configuration réseau ou de politiques de sécurité.

Une exploitation dynamique et mixte d'une grappe telle que nous venons de la décrire, engendre des contraintes d'exploitation supplémentaires. Les principales caractéristiques recherchées sont les suivantes :

- le déploiement rapide (en quelques minutes) sur un grand nombre de nœuds (plusieurs dizaines) de la configuration logicielle adéquate pour un utilisateur donné.
- l'isolation des ressources allouées à différents utilisateurs, c'est à dire fournir les garanties nécessaires pour que les propriétés de sécurité et les performances fournies soit équivalentes à celles d'une grappe à usage exclusif.
- l'abstraction des ressources allouées. Pour simplifier l'administration et l'exploitation d'une grappe partagée, il est souhaitable de manipuler des abstractions de plus haut niveau qu'une liste de nœuds réservés à un utilisateur. Ceci motive l'idée d'allouer des grappes virtuelles avec les propriétés d'isolation définies ci-dessus. De plus, la taille d'une grappe virtuelle doit pouvoir varier dynamiquement en fonction des contraintes de charge et de la politique d'allocation des ressources.

Des infrastructures telles que *Cluster On Demand* [152, 47] développée à Duke University commencent à répondre à ces besoins. D'autres approches récentes vont plus loin en utilisant des nœuds virtuels encapsulés dans des machines virtuelles (émulant un environnement physique complet), ce qui permet de renforcer l'isolation entre différents services hébergés sur un même nœud, de découpler la topologie applicative des ressources physiques utilisées et d'influer dynamiquement sur cette dernière par des mécanismes de migration (intégrés à la machine virtuelle) efficaces et non intrusifs vis-à-vis du système d'exploitation [116, 36].

1.3 Technologies d'interconnexion de machines

Comme déjà évoqué précédemment, le réseau d'interconnexion d'une grappe a un impact important sur ses performances. Les premières grappes, construites à partir de PC standards, ont tout naturellement été basées sur un réseau Ethernet, qui équipait (et équipe toujours) une grande majorité des PC et stations de travail disponibles. L'emploi de ce type de matériel a été couplé avec celui de piles protocolaires (logicielles) classiques comme TCP/IP. Ces protocoles furent initialement développés pour fournir des canaux virtuels « sans erreur » au dessus de réseaux hétérogènes, à grande échelle, non fiables, de topologie variable et complexe. Ces contraintes ne s'appliquent pas ou peu au contexte des grappes de machines alors que leur prise en compte dans les protocoles pénalise les performances d'un réseau (latence et bande passante). Partant de ce constat, de nouvelles technologies pour réseaux locaux ont été développées, mettant l'accent sur les performances, en traitant notamment les erreurs de transmission de façon matérielle et en s'affranchissant de nombreuses contraintes de distance et de routage.

Cette section commence par présenter l'état actuel de la technologie Ethernet puis aborde les principaux exemples de réseaux optimisés[4] déployés aujourd'hui. Enfin, leurs perspectives d'évolution proches sont évoquées.

[4]Ces réseaux sont parfois désignés par l'expression « réseaux à hautes performances » mais l'évolution de la technologie Ethernet rend cette distinction moins lisible.

1.3.1 Réseaux Ethernet

Le réseau Ethernet, développé dans les laboratoires de Xerox au début des années 1970 et normalisé dans les années 1980, est la technologie pour réseau local la plus répandue, offrant un rapport performance/coût très avantageux.

Des équipements à la norme FastEthernet (qui offrent un débit théorique de 100 Mbits/s et un débit réel compris entre 80 et 95 Mbits/s) équipent aujourd'hui toutes les machines d'entrée de gamme. Les répéteurs (ou *hubs*) sont tombés en désuétude et ont été remplacés par des commutateurs qui limitent les problèmes de collisions en effectuant du routage au niveau des adresses MAC (*Medium Access Control*). La quasi-totalité des grappes actuelles sont équipées d'un réseau FastEthernet. Dans le cas des grappes bon marché, constituées de PC de bureau, il s'agit de l'unique réseau de la grappe. Dans le cas des grappes plus performantes, il sert à isoler le trafic des commandes d'administration de celui des données applicatives.

La technologie Gigabit Ethernet a déçu lors de son introduction, en offrant de faibles performances (surtout au niveau des latences) pour un prix comparable à celui d'un réseau optimisé. Ce n'est plus le cas aujourd'hui : les prix des interfaces ont baissé[5] et les performances se sont sensiblement améliorées (entre 700 et 940 Mbits/s de débit réel, et une latence de 20 à 30 μs) notamment grâce à l'intégration de contrôleurs Ethernet directement sur les cartes mères. Gigabit Ethernet permet également d'augmenter la taille des paquets (auparavant limitée à 1500 octets) jusqu'à 9000 octets et d'augmenter ainsi les performances de gros transferts de données. L'utilisation des *jumbo frames* est cependant peu fréquente car elle nécessite un commutateur gigabit administrable, dont le prix reste élevé.

Comme nous l'avons déjà évoqué dans l'introduction de cette section, les réseaux Ethernet sont intimement associés aux piles protocolaires TCP/IP et UDP/IP. L'avantage de ces protocoles réside dans leur robustesse (au niveau de leur tolérance aux pannes intrinsèque aussi bien qu'à celui de la maturité de leurs implémentations) ainsi que dans la popularité de leur interface de programmation par *sockets*, simple et portable. Ces piles induisent néanmoins un surcoût important, tant au niveau des communications, qu'à celui des applications. Certains projets de recherche, tels Gamma [53] ont essayé d'utiliser des réseaux Ethernet sans recourir aux protocoles classiques, mais il n'est pas trivial d'améliorer significativement les performances, car une gestion logicielle (même simplifiée) des potentielles erreurs de communication s'avère toujours nécessaire. C'est ce qui a motivé la conception de réseaux locaux spécialisés, gérant les erreurs de communication de façon matérielle.

1.3.2 Réseaux spécialisés

Les années 1990 ont vu apparaître de nombreux réseaux pour grappes, offrant des performances nettement supérieures à celles de la famille Ethernet. Le prix du matériel est cependant bien plus élevé (environ $1000 pour une interface réseau optimisée contre $100 pour une carte Gigabit Ethernet). Il est à noter que la plupart de ces réseaux peuvent être utilisés avec une pile protocolaire TCP/IP ou UDP/IP optimisée mais le rapport performance/prix de ce mode d'utilisation est limité et ne motive généralement pas à lui seul l'emploi d'un tel réseau (d'autant plus que, contrairement à Ethernet, le multicast matériel est émulé par une boucle d'envoi, ce qui pénalise les performances des diffusions).

Cette section commence par expliciter la technique d'accès direct à de la mémoire distante (RDMA), utilisée par la plupart des technologies spécialisées actuelles. Elle présente ensuite deux des réseaux les plus utilisés aujourd'hui, SCI et Myrinet, dont les cartes se connectent au bus PCI des machines. Elle évoque enfin Infiniband, une architecture d'interconnexion unifiée pour nœuds,

[5] Seuls les commutateurs de grande taille (> 50 ports) restent chers, mais demeurent plus abordables que les commutateurs de même taille pour réseau optimisé, avec toutefois des latences plus élevées

réseaux et périphériques de stockage.

1.3.2.1 Accès direct à une mémoire distante (RDMA)

Comme son nom l'indique, la technique d'accès direct à une mémoire distante (*Remote Direct Memory Access*) correspond à un extension du principe local de DMA. Le DMA permet d'effectuer directement des transferts entre un contrôleur de périphérique et la mémoire centrale d'un nœud. L'intérêt principal de cette méthode tient au fait qu'elle décharge le processeur central de la gestion complète des transferts. Ce dernier est alors seulement impliqué lors de l'amorçage et de l'acquittement du transfert.

La technique de RDMA permet un transfert direct entre les mémoires de deux nœuds sans intervention du processeur distant (et avec les même étapes d'amorçage et d'acquittement au niveau du processeur de l'émetteur).

1.3.2.2 Scalable Coherent Interface (SCI)

La technologie SCI est issue d'une norme IEEE de 1992 [111] mais il n'existe aujourd'hui qu'un seul fournisseur de périphériques : le norvégien Dolphin Interconnect Solutions. SCI supporte deux modèles de communication : mémoire partagée et passage de messages. Sa principale originalité est sa capacité d'adressage : elle permet au processeur d'un nœud d'accéder, via ses classiques instructions *load/store*, à la mémoire d'un nœud distant sans interrompre le(s) processeur(s) de ce dernier. Cette caractéristique a notamment été exploitée pour implémenter des mémoires partagées distribuées logicielles efficaces [42]. En complément des entrées/sorties programmées, les cartes SCI permettent également de décharger les processeurs des transferts en utilisant un moteur de RDMA aussi bien en lecture qu'en écriture.

Les cartes SCI fournissent de très bonnes performances, aussi bien en latence (1,5 μs, parmi les plus faibles) qu'en débit (jusqu'à 280 Mo/s de débit réel sur des architectures 32 bits et 320 Mo/s sur des architectures 64 bits). La topologie de base d'un réseau SCI est l'anneau. Il existe des cartes SCI multidimensionnelles (2D ou 3D) qui permettent de relier un même nœud à plusieurs anneaux pour obtenir plus de tolérance aux fautes. Il est aussi possible d'utiliser un commutateur pour relier directement plusieurs nœuds ou plusieurs anneaux entre eux. Il n'existe cependant pas de commutateur offrant plus de 8 ports pour le moment.

1.3.2.3 Myrinet

Myrinet [27] est une technologie issue des travaux des projets Mosaic et Atomic LAN (Caltech/USC), commercialisée par la société Myricom, et normalisée par l'ANSI. Les cartes Myrinet sont équipées d'un processeur RISC, le LANai et de quelques Mo de mémoire vive (processeur à 330Mhz et 4 Mo de mémoire pour les modèles les plus récents, avec une latence de 5,7 μs et un débit maximum de 495 Mo/s). Myrinet est basé sur une architecture entièrement commutée et sur une communication par passage de messages. La taille des messages n'est pas limitée, ce qui rend le réseau très performant pour des messages de grande taille. Les messages de petite taille sont pénalisés par le temps d'armement du DMA mais les performances restent bonnes. Le logiciel GM fourni avec les cartes permet au code applicatif de communiquer directement avec une carte Myrinet sans passer par le système d'exploitation (*OS bypass*) et permet ainsi d'éviter de coûteuses copies de données. Enfin, il est possible de reprogrammer le microprogramme (*firmware*) du LANai pour modifier le protocole de transport pour améliorer les performances en fonction des contraintes applicatives [179].

1.3.2.4 Infiniband

Infiniband (ou IBA) [173] est un standard industriel récent (2000) qui définit une nouvelle architecture d'entrées/sorties, unifiée pour différents aspects : interconnexion de nœuds, de réseaux et de périphériques. L'objectif principal d'IBA est d'éviter les lacunes des architectures d'interconnexion classiques basées sur une structure en bus (la plus utilisée étant actuellement le bus PCI) : nécessité de coopération entre les différents périphériques pour le partage de l'espace d'adressage et de la bande passante, difficulté voire impossibilité pour le système de détecter et d'isoler un périphérique défaillant.

IBA repose sur un modèle entièrement commuté. Les transferts de données entre deux entités communicantes sont donc effectués selon un mode point à point, sous la forme de messages. Un message peut correspondre à différents types d'opérations dont notamment un envoi de données sur un canal connecté, un datagramme, un accès direct à une mémoire distante (RDMA), une opération atomique en mémoire distante, une communication multicast. Des services de communication optionnels sont également prévus, dont un mode de « datagrammes fiables », qui permet de multiplexer un même canal fiable entre plusieurs contextes de communication répartis sur une même paire de nœuds.

La spécification IBA distingue les équipements à destination des périphériques (Target Channel Adapter — TCA) et ceux à destination des nœuds (*Host Channel Adapter* — HCA). Un TCA implémente les fonctions de base du protocole et interopère avec le périphérique qu'il raccorde au réseau. Un HCA doit, en plus de l'implémentation matérielle du protocole de liaison de données, être relié au contrôleur mémoire du nœud et exporter une interface de programmation au système. La façon dont le contrôleur mémoire et le HCA sont interconnectés n'est pas spécifiée de manière précise. Ainsi, les HCA disponibles jusqu'à présent ne sont rien d'autres que des cartes réseau « traditionnelles », qui communiquent avec la mémoire par l'intermédiaire du bus PCI. Il ne s'agit cependant que d'une étape transitoire, dans l'attente de mises en œuvre utilisant Infiniband comme mécanisme d'interconnexion central et unifié. Les HCA seront alors directement reliés au contrôleur mémoire. Les cartes Infiniband disponibles actuellement sont capables d'OS-bypass et fournissent des performances de l'ordre de 6 μs et 800 Mo/s en débit.

Enfin, IBA repose sur le schéma d'adressage à 128 bits d'IPv6 pour identifier chaque HCA. Ceci permet d'envisager des architectures à grande échelle interconnectant de nombreux nœuds, périphériques et réseaux par l'intermédiaire de commutateurs et de routeurs.

1.3.3 Perspectives

Il est assez difficile de formuler un pronostic précis sur l'évolution à moyen terme des technologies d'interconnexion et surtout au sujet de leur succès commercial.

Les réseaux Ethernet sont omniprésents et vont probablement continuer à dominer les installations industrielles étant donné le coût acceptable et la transition technologique simple qu'ils permettent lors de l'introduction d'une nouvelle famille de matériel. La généralisation de l'Ethernet à 10 Gbit/s est d'ailleurs annoncée dans un futur proche. Le problème principal d'Ethernet demeure plus que jamais le coût associé aux piles protocolaires TCP/IP dont l'impact devient de plus en plus prépondérant au niveau des performances. Plusieurs pistes sont actuellement explorées pour résoudre ce problème.

- La première vise à confier la gestion des protocoles coûteux à un processeur spécialisé, embarqué au sein d'une carte réseau (*protocol offloading*). Cette méthode est sujette à d'importantes controverses quant aux gains qu'il est possible d'envisager [150, 202].
- La seconde idée (*protocol onloading*) est à l'opposé de la précédente. Elle consiste à profiter des nouvelles architectures de processeurs CMP (cf. 1.2.1) et à développer des optimisations

basées sur le postulat qu'un ou plusieurs cœurs sont consacrés exclusivement au traitement des protocoles réseau [184].

- La dernière approche est compatible avec les propositions précédentes. Elle repose sur le développement de mécanismes de RDMA adaptés à la technologie Ethernet.

Ces différentes optimisations n'en sont, à l'heure actuelle, qu'au stade de prototype. Les problèmes actuels sont donc amenés à durer, au moins à moyen terme. De plus, il n'est pas évident que l'introduction de ces nouvelles caractéristiques n'ait pas un coût élevé, qui pourrait nuire au ratio performance/prix qui a fait le succès de la gamme Ethernet jusqu'à présent.

Les interfaces spécialisées demeurent la solution de choix pour les applications contraintes par des paramètres tels que la latence et la consommation CPU associée aux communications[6]. Les perspectives dans le domaine des réseaux optimisés pour grappes sont cependant assez confuses.

Il existe un nombre significatif de technologies distinctes (SCI, VIA, Myrinet, Quadrics, Infiniband, etc.). Si la plupart d'entre elles partagent des caractéristiques communes (fiabilité des communications RDMA, OS-bypass), leurs mises en œuvre sont très différentes et ne sont pas intéropérables. Il est difficile de déterminer si toutes ces technologies vont continuer à cohabiter au cours des années à venir ou si la transition architecturale majeure proposée par Infinband va finir par l'emporter.

L'écriture d'applications à la fois optimisées pour le réseau sous-jacent et portables facilement sur différentes plates-formes matérielles est un problème complexe. Il est toujours possible de trouver un « dénominateur commun » pour l'interface de programmation des communications (par exemple, l'API usuelle de *sockets* dont la plupart des fabricants fournissent une version adaptée à leur matériel) mais le coût des couches de protocoles additionnelles peut dans certains cas être jugé prohibitif par rapport aux capacités du réseau.

Le problème d'une interface de programmation commune se pose aussi à bas niveau, y compris pour une technologie donnée. La spécification Infiniband ne définit ainsi pas d'interface de programmation concrête mais uniquement une description verbale des fonctionnalités à implémenter. Il existe donc différentes API relatives au mêmes notions. Dans un cadre plus général, et malgré des efforts en cours, il n'existe pas non plus d'API de référence pour la description des transferts par RDMA (dont les grands principes sont pourtant communs aux différentes technologies).

1.4 Bilan

Ce chapitre a présenté un bref survol du domaine des grappes de machines. Nous tentons maintenant de dresser les enseignements importants qui s'en dégagent, du point de vue des programmeurs de systèmes d'exploitation et d'intergiciels pour ces plates-formes.

L'émergence des serveurs de données en grappe introduit une nouvelle hiérarchie de contraintes où les aspects de gestion dynamique des ressources et de tolérance aux pannes priment autant que les performances.

L'architecture des microprocesseurs équipant les nœuds d'une grappe prend actuellement un virage majeur. Ceux-ci vont fournir de plus en plus de support matériel pour le parallélisme d'exécution. L'accélération des applications va donc passer par un recours croissant aux techniques de programmation parallèle. Cependant, toutes les tâches applicatives ne sont pas parallélisables à l'infini. L'augmentation de la réactivité d'un environnement en grappe va donc passer par une utilisation maximale des ressources matérielles et inciter à une restructuration des systèmes d'exploitation et intergiciels au

[6] La présence de processeurs embarqués généralistes peut en outre être exploitée pour déporter certains traitements applicatifs [231].

niveau de certaines opérations critiques comme l'ordonnancement, les communications et le stockage de données.

En outre, même si les principes de communication employés par les périphériques réseau modernes sont clairement identifiés, il existe encore de nombreux problèmes d'hétérogénéité au niveau des interfaces de programmation existantes. Dans l'attente de consensus et de véritables efforts de normalisation, il est important de (continuer à) développer des infrastructures logicielles permettant le développement d'applications génériques, indépendantes du matériel, tout en laissant la porte ouverte à des optimisations spécifiques à bas niveau.

Chapitre 2

Technologies de stockage de données

Sommaire

2.1 Introduction

Ce chapitre s'intéresse aux technologies matérielles et logicielles permettant de stocker et d'accéder à des données. L'introduction de ce chapitre se limite au cas d'un système centralisé ; les autres sections étendront les notions présentées au contexte de systèmes distribués de type grappes de machines.

Cette section commence par faire le point sur les principaux supports matériels utilisés pour assurer la persistance des données, c'est à dire la faculté de conserver des informations de manière durable, même en cas d'arrêt (logique et électrique) d'un système. Nous présentons ensuite les différentes interfaces associées aux couches qui composent un système de stockage.

2.1.1 Supports physiques pour la persistance

Aujourd'hui, les principales technologies matérielles utilisées pour garantir la persistance de données numériques sont les technologies optiques (CD, DVD ...), les bandes magnétiques, les mémoires non volatiles et les disques durs.

Les supports optiques et les bandes magnétiques sont surtout utilisés pour l'archivage de données, d'une part en raison de leur bon ratio densité/prix, d'autre part car certaines caractéristiques ne les prédisposent pas à un autre type d'utilisation plus intensif. Les temps d'accès sont longs et seul un accès séquentiel est possible (en écriture pour les supports optiques) ou viable[1].

Les mémoires non volatiles[2] sont fabriquées à partir de matériaux semiconducteurs. Certaines mémoires non volatiles (EEPROM, FLASH) ont des temps d'écriture longs et se dégradent après quelques millions d'écritures ; elles sont surtout employées pour des applications peu critiques (telles que les appareils photo numériques) ou embarquées (carte à puce). D'autres mémoires non volatiles (principalement de la SRAM, similaire à celle utilisée dans les caches, couplée à une batterie pour être non volatile) offrent de bien meilleures performances mais pour un coût élevé. De ce fait, elles ne sont pour l'instant utilisées que dans le cas de systèmes critiques (par exemple pour le journal des opérations à effectuer ou annuler pour maintenir un état cohérent), et ne stockent généralement que de faibles volumes de données (quelques mégaoctets).

Les disques durs offrent le meilleur compromis en termes de densité/performances/mode d'accès (aléatoire en lecture et en écriture comme pour les mémoires)/fiabilité. Un disque dur est un empilement de plusieurs plateaux recouverts d'une couche d'oxyde de métal magnétique. Les plateaux sont maintenus par un axe central qui les fait tourner à vitesse constante. Une tête de lecture-écriture est associée à chaque face d'un plateau. Sur chaque plateau, les données sont organisées par pistes, chaque piste étant divisée en secteurs de taille constante (512 octets généralement[3]). La connaissance de la « géométrie interne » d'un disque n'est pas nécessaire pour dialoguer avec celui-ci : elle est abstraite par le contrôleur interne du disque, qui présente l'espace de stockage comme un ensemble linéaire de secteurs[4]. Outre les bénéfices en termes de généricité et de simplification des pilotes de disques, ceci permet également au contrôleur interne de reloger certains secteurs défectueux à un autre emplacement (quelques secteurs de rechange étant prévus à cet effet). Un niveau d'abstraction supplémentaire consiste à diviser un disque en plusieurs partitions logiques, isolées entre elles.

Le disque dur est bientôt cinquantenaire. Le premier disque, l'IBM RAMAC apparut en 1956 ; il permettait de stocker 5 Mo et offrait un débit de 8 ko/s. Les progrès, en matière de densité de stockage, de temps d'accès et de débit, ont été incessants. A partir de 1989, la densité des disques a doublé chaque année[5]. Le coût relatif du stockage a également considérablement baissé (2 $/Mo en 1992 et 2 $/Go en 2002). Il n'en est pas de même pour les temps d'accès et les débits : les très fortes contraintes électro-mécaniques (mouvement du bras de lecture/ecriture, vitesse de rotation du disque) n'ont permis qu'une amélioration d'un facteur 10 pour les temps d'accès et 40 pour les débits [86]. Une rétrospective détaillée sur l'évolution des disques durs et l'impact de cette technologie est présentée dans [89]. Les tableaux 2.1 et 2.2 inspirés d'une présentation de Garth Gibson [82], résument et projettent l'évolution des disques dans le passé et le futur proches.

On peut tirer plusieurs conclusions significatives de l'évolution des caractéristiques des disques durs. Tout d'abord, il existe un fossé croissant entre les performances d'un disque et celle d'une

[1] Les lecteurs de bandes sont capables d'accès aléatoires mais avec des performances 1000 fois inférieures à celles d'un disque dur.

[2] NVRAM en anglais pour *Non Volatile Random Access Memory*

[3] Cette taille est fixée lors du *formatage de bas niveau* du disque.

[4] Cette interface est généralement désignée par l'expression *Logical Block Addressing* ou LBA.

[5] Soit 100% d'augmentation annuelle, à comparer aux 60% annuels de la loi de Moore.

Disque dur 3,5 pouces	1992	2002	2012 (?)
Capacité	320 Mo	100 Go	30 To
$/octet	2/M	2/G	2/T
Taille d'une piste (ko)	32	330	3400
Temps d'accès (ms)	20	10	5
Débit en lecture (Mo/s)	1,7	40	800
Temps total de lecture	3 min	40 min	11 h
Vitesse des réseaux locaux (bit/s)	10 M	1 G	100 G

TAB. 2.1 – Rétro et prospective sur l'évolution récente des disques durs

Matrice de disques	1992	2002	2012 (?)
Capacité	22 Go	28 To	36 Po
Entrées-Sorties / s	600	3000	20000
Débit (Mo/s)	1,7	40	800
Temps total de lecture	24 min	41 h	24 h

TAB. 2.2 – Rétro et prospective sur l'évolution des matrices de disques durs

unité centrale (processeur(s) et mémoire centrale). Les Entrées/Sorties (E/S)[6] sont donc un goulot d'étranglement de plus en plus critique pour les performances des machines. D'autre part, la densité et la capacité des disques augmentent très rapidement, et en particulier :

- beaucoup plus vite que les performances de lecture/écriture. Par conséquent, le temps d'accès à la totalité des informations stockées sur un disque croit d'année en année. Pour limiter au maximum cet état de fait et optimiser les performances, les accès aux données devront être très majoritairement séquentiels [86] (les accès aléatoires offrent des débits bien plus faibles car ils sont pénalisés par le temps de positionnement de la tête de lecture). Le recours à des techniques optimisées pour cet objectif, telles que les *write-anywhere file systems* [102] ou les *log-structured file systems* [188] (cf 2.1.2.3), risque de se généraliser dans les années à venir.

- sensiblement plus vite que la plupart des besoins en volume de stockage de données numériques. Ceci incitera probablement à ne plus utiliser les disques uniquement comme mémoire secondaire, mais également comme mémoire d'archivage. En outre, les techniques de redondance de données basées sur des sommes de contrôle ou des codes correcteurs d'erreurs seront peut être moins prisées, au profit de techniques de duplication complète[7], plus brutales (coûteuses en place) mais également plus simples à mettre en œuvre et plus efficaces.

Enfin, on peut également constater un fossé croissant entre les performances (débit, latence) offertes par un disque et celles fournies par une interface réseau. Ceci motive le déploiement d'infrastructures de stockage distribuées, où la capacité de transfert d'une seule carte réseau équipant un nœud client permet d'absorber le débit de nombreux disques distants.

[6] Dans la suite de ce document, nous n'employons les termes « Entrées/Sorties » ou « E/S » que pour les opérations liées au stockage de données et/ou aux communications en réseau. Les interactions avec d'autres types de périphériques (capteurs, commande de processus industriels...) ne sont pas incluses dans cette désignation.

[7] C'est à dire privilégier des approches de type RAID 1 ou 1+0 par rapport à celles de type RAID 5 (voir section 2.1.2.2).

2.1.2 Différentes interfaces

Cette section présente les différents niveaux d'interface impliqués dans la gestions des E/S, en commençant par les couches logiques les plus basses. L'accès au médium physique de stockage ne sera pas abordé.

2.1.2.1 Niveau « contrôleur »

Il s'agit du plus bas niveau logique qui permet le dialogue entre un contrôleur (intégré à la carte mère ou interconnecté à celle-ci via le bus PCI) et un périphérique de stockage (reliés par un bus). Nous nous limitons au contexte des disques durs, où il existe deux principales interfaces[8] : IDE (Integrated Device Electronics)[9] et SCSI (Small Computer System Interface).

Outre le dialogue nécessaire à l'initialisation du système, les commandes échangées entre le contrôleur et le périphérique sont rudimentaires ; il s'agit d'ordres de lecture et d'écriture élémentaires d'un ou plusieurs secteurs adjacents. Les transferts de données sont basés sur l'utilisation d'un DMA.

L'interface SCSI est plus complexe que l'interface IDE et offre plus de possibilités, notamment aux niveaux :

- du nombre de périphériques interconnectés (15 pour SCSI contre 2 pour l'IDE)
- du débit du bus (160 à 320 Mo/s en SCSI contre 133 à 150 Mo/s pour l'IDE)
- de l'« intelligence des périphériques » : les périphériques SCSI peuvent gérer jusqu'à 256 commandes en attente (et les réordonnancer pour optimiser les accès au médium physique alors qu'un périphérique IDE ne peut accepter qu'une seule commande à la fois (ou un faible nombre pour certaines implémentations). L'interface SCSI permet également une meilleure gestion de la concurrence des accès au bus ainsi qu'une gestion hiérarchique des entités qui y sont reliées[10].

Pour ces raisons, les disques « haut de gamme » ont traditionnellement été basés sur l'interface SCSI et l'IDE a occupé le reste du marché. Bien plus que l'interface externe, ce sont en fait les conséquences de cette segmentation de marché sur l'architecture interne des disques qui a un impact majeur sur les performances observées [9].

Ceci étant dit, des études [234] ont montré que, contrairement aux idées reçues,

- les performances des disques IDE et SCSI sont proches
- les disques IDE sont généralement plus rapides que les disques SCSI pour les accès séquentiels mais sont effectivement plus lents pour les accès aléatoires
- le handicap des disques IDE au niveau des accès aléatoires peut être réduit à des proportions acceptables en utilisant des techniques d'optimisation de requêtes au niveau du système d'exploitation.

Les disques SCSI prédominent encore actuellement mais une part croissante des serveurs sont équipés de disques IDE, beaucoup moins chers. Les matrices RAID[11] à base de disques IDE sont

[8] Ces interfaces ne sont toutefois pas spécifiques aux disques durs. Elles sont utilisés pour d'autres types de périphériques, par exemple les graveurs de CD-ROM.

[9] les terminologies ATA (AT-Attached), EIDE (Enhanced IDE), SATA (Serial ATA) sont des évolutions à associer à la norme IDE.

[10] Cette gestion hiérarchique est mise en œuvre grâce à la notion de « sous-unités » (LUN pour *Logical Unit Number*) associées à un identifiant sur le bus SCSI. Ce principe est par exemple appliqué aux matrices de disques : le contrôleur d'une matrice est désigné par un identifiant SCSI et le LUN 0 et chacun des disques est associé à un autre LUN.

[11] Les techniques RAID sont abordées en section 2.1.2.2

également de plus en plus populaires. L'émergence de la norme SATA [88] (une implémentation série du protocole qui permet notamment des communications points à points et des débits plus élevés) amplifiera sans doute cet état de fait.

Il convient également de préciser que certains systèmes (dont Linux) fournissent un pilote générique SCSI qui permet à des périphériques « non SCSI » (par exemple IDE ou USB) d'exporter une interface SCSI. Ceci est utile pour présenter une interface unique au niveau des couches basses, par exemple

- pour les logiciels particuliers (gravure de CD, acquisition d'images,...) qui accèdent directement à une interface de niveau contrôleur (voir section 2.1.3) ;
- pour le transport réseau de commandes de niveau contrôleur (voir section 2.2).

Pour simplifier, abstraire et enrichir la sémantique de la manipulation de données, des interfaces de plus haut niveau que celle du contrôleur sont couramment utilisées. Celles-ci sont décrites dans les section suivantes.

2.1.2.2 Niveau « blocs »

Le niveau blocs n'enrichit pas ou peu la sémantique du niveau contrôleur : il définit également un espace linéaire de blocs de taille fixe (supérieure ou égale à la taille des secteurs du périphérique sous-jacent) et des opérations élémentaires de lecture et d'écriture d'un ou plusieurs blocs adjacents. L'intérêt principal d'un *périphérique (générique) à blocs* (PGB ou *block device* en anglais) est de fournir une abstraction uniforme de l'espace de stockage, indépendante des caractéristiques (au niveau contrôleur) des périphériques sous-jacents.

La taille des blocs varie en fonction des périphériques concernés (et il est souvent possible de la modifier pour un périphérique donné). Dans le cas des disques, cette taille est généralement fixée à 4 ko (ce qui correspond à la taille des pages mémoire de la plupart des systèmes d'exploitation). Certaines applications particulières telles que les bases de données utilisent des tailles de blocs beaucoup plus grandes (jusqu'à 64 ko) pour optimiser les accès séquentiels.

Dans le cas le plus simple, un PGB correspond au pilote du matériel sous-jacent (par exemple, un pilote de disque IDE ou SCSI), mais plusieurs couches de niveau blocs peuvent également être empilées. En voici deux exemples représentatifs :

RAID L'idée du RAID (acronyme de *Redundant Array of Independent Disks*[12]) [169, 50, 143] est d'appliquer les concepts généraux de parallélisme et de redondance aux disques durs afin d'améliorer leurs performances et leur fiabilité tout en conservant l'abstraction d'un disque unique. Plusieurs organisations logiques ont été définies et nommées par des numéros ; on parle de *niveaux de RAID*, mais le terme est mal choisi car il n'y a pas de hiérarchie entre ces différentes configurations ; un niveau donné correspond à un compromis particulier en termes de tolérance aux fautes, de performances et de coût (nombre de disques nécessaires).

L'amélioration des performances est réalisée en distribuant les données sur plusieurs disques de façon à effectuer des accès parallèles [191]. Les données sont découpées (on parle de *striping*) en morceaux (*chunks*) de taille fixe répartis sur les différents disques. L'ensemble des morceaux constituant une ligne de la matrice est dénommé *bande* (*stripe* en anglais). La taille d'un morceau s'exprime en nombre de blocs (dont la taille est définie par le PGB de type RAID). Il n'y a pas de paramétrage universel d'une matrice RAID ; la taille idéale des morceaux est très dépendante du type de données stockées (c'est-à-dire de leurs tailles et des profils d'accès à ces

[12] qui fut préféré par l'industrie à celui initialement choisi : *Redundant Array of Inexpensive Disks*. Aujourd'hui, un système RAID ne permet pas (plus) de diminuer le coût d'un système de stockage, c'est même le contraire.

données). Les tailles de morceau généralement choisies varient de quelques dizaines à quelques centaines de blocs.

L'amélioration de la fiabilité est basée sur deux techniques de redondance :

- soit en copiant intégralement les données sur un ou plusieurs disques *miroirs* (RAID 1),
- soit en stockant un *morceau de parité* (calculé avec des OU-exclusifs) pour chaque ligne de la matrice (cas du RAID 5).

La figure 2.1 illustre les niveaux RAID les plus utilisés

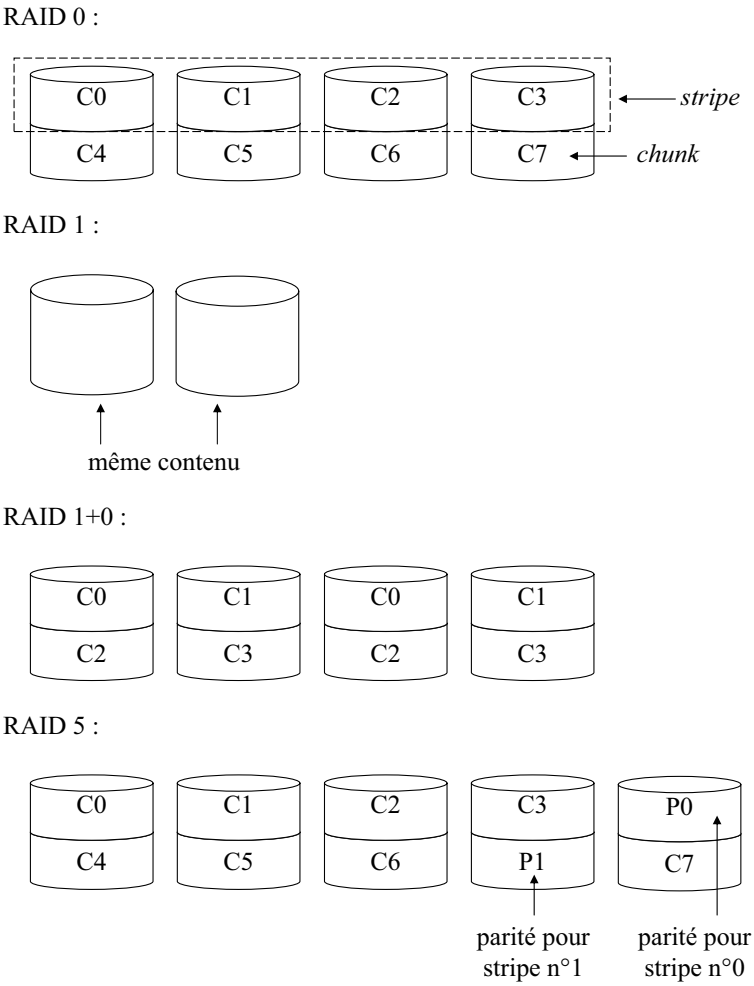


Figure 2.1 – Principaux schémas de répartition de données sur disques (RAID)

Le niveau RAID 0 est un cas particulier car il n'utilise pas de redondance et n'est donc pas tolérant aux pannes. Il est utilisé pour stocker très efficacement des données peu critiques.

Le RAID 1 est coûteux de par sa redondance brutale, mais fournit de bonnes performances (supérieures à celle d'un disque local pour les lectures et légèrement inférieures pour les écritures).

Le RAID 5 entrelace données et informations de parité au sein de la matrice. La redondance par parité le rend moins coûteux en espace de stockage que le RAID 1. Les performances sont cependant moins bonnes, surtout pour les écritures et la reconstruction d'une matrice (en cas

de panne d'un disque) est complexe[13].

Le RAID 1+0 (ou 10)[14] est une combinaison des niveaux 0 et 1, il offre un bon niveau de protection (plusieurs pannes peuvent être tolérées dans certaines circonstances) et fournit de bien meilleures performances qu'un disque unique.

Pour un niveau RAID donné, le nombre de disques (c'est-à-dire la taille d'une ligne de la matrice) peut être variable ; on peut par exemple implémenter une matrice RAID 5 avec un nombre quelconque de disques (supérieur à 2). Pour les niveaux de RAID avec parité, augmenter le nombre de disques de la matrice permet de diminuer le surcoût dû à la redondance des données. Cependant utiliser un nombre élevé de disques présente plusieurs désagréments :

- la probabilité de pannes augmente avec le nombre de disques.
- pour le RAID avec parité, les grandes matrices souffrent de mauvaises performances en écriture et en reconstruction.

Pour ces raisons, les matrices RAID sont généralement restreintes à des tailles de l'ordre de 4 à 10 disques.

Dans le cas d'un contrôleur RAID matériel, le système n'utilise qu'une couche logicielle de niveau bloc (le pilote du contrôleur) ; dans le cas d'un RAID émulé de façon logicielle, une (ou plusieurs, par exemple dans le cas du RAID 1+0) couche(s) RAID est (sont) empilée(s) sur la couche bloc des pilotes concernés. Les émulations logicielles sont généralement cantonnées aux niveaux 0, 1 et 1+0 car pour les niveaux avec parité, les performances sont moindres que celles d'un système matériel et les calculs de parité pèsent d'un poids non négligeable sur les performances du système.

Gestionnaires de volumes logiques Le rôle d'un gestionnaire de volumes logiques (GVL ou LVM en anglais pour *Logical Volume Manager*) [221] est d'agréger la capacité de stockage de plusieurs périphériques blocs pour présenter une abstraction unifiée aux couches plus hautes du système. Ce rôle n'est pas éloigné de celui des systèmes RAID logiciels et certains GVL intègrent d'ailleurs ce type de fonctionnalité[15]. Un GVL présente cependant un intérêt complémentaire : celui de rajouter un niveau d'indirection entre un périphérique et la couche supérieure qui l'utilise[16]. Ceci permet notamment de reconfigurer dynamiquement la composition d'un volume logique. Par exemple, un volume peut initialement être associé à une unique partition et si l'espace vient à manquer, d'autres partitions (issues du même disque ou bien d'un ou de plusieurs autres) peuvent êtres greffées au volume logique, sans interrompre le fonctionnement des couches supérieures[17].

2.1.2.3 Niveau « fichiers »

L'interface de niveau « fichiers », est plus riche que celle des blocs. Elle fournit une vue logique plus pratique pour les utilisateurs du système d'exploitation (qu'il s'agisse d'applications ou d'humains). Un fichier est une collection nommée d'informations apparentées, enregistrée sur un stockage

[13] Ce qui se traduit par une grande fenêtre temporelle pendant laquelle la matrice n'est plus tolérantes aux fautes et les performances sont dégradées.

[14] Les premières publications sur le RAID [169] englobaient cette topologie dans le niveau RAID 1. L'industrie a cependant explicité la différence entre deux (ou n) disques miroirs (ce que nous avons nommé RAID 1) et la combinaison de techniques de striping et mirroring (RAID 1+0). Il existe également la combinaison inverse, RAID 0+1, (mirroring au dessus de disques strippés), qui présente des performances proches mais moins de tolérance aux pannes.

[15] Dans certains cas, un GVL peut être combiné à des périphériques RAID matériels. Par exemple, deux matrices RAID 1 peuvent être agrégées par du striping au niveau du GVL, aboutissant ainsi à une topologie RAID 1+0.

[16] Cette couche supérieure est généralement un système de fichiers (voir section 2.1.2.3).

[17] De façon similaire, des partitions inutilisées peuvent être retirées de la composition d'un volume.

secondaire ; sa taille peut en général être variable (à la création et au cours du temps). Il est possible de classer les fichiers dans une hiérarchie de répertoires (eux mêmes implémentés comme des fichiers particuliers). Le rôle d'un *système (de gestion) de fichiers* (SGF[18]) est notamment de gérer la correspondance entre un fichier et les blocs du périphérique sous-jacent qui lui sont associés. On parle de méta-données, pour désigner les informations que le SGF stocke à cet effet.

Les principales opérations permises par un SGF sont

- la création/suppression d'un fichier,
- la lecture/écriture dans un fichier,
- le repositionnement dans un fichier,
- la troncature d'un fichier,
- le déplacement/renommage d'un fichier,
- la gestion des attributs d'un fichier (droit d'accès, date de modification, taille,...).

Un SGF assure en outre la cohérence du contenu d'un fichier dans le cas d'accès concurrents. Pour cela, il est nécessaire de synchroniser les accès à un fichier par l'emploi de verrous. La plupart du temps, le niveau de granularité du verrouillage est le fichier complet. Le schéma le plus courant est d'autoriser à un instant donné soit un ou plusieurs accès en lecture, soit un accès unique en écriture. Pour certaines applications (de calcul scientifique, par exemple), plusieurs processus ont besoin d'écrire simultanément dans des zones distinctes d'un même fichier. Un verrouillage de grain plus fin est alors employé, en général par plages d'octets (*byte range locking*).

Un SGF implémente également une sémantique de cohérence particulière, qui définit un ensemble de garanties concernant la vue qu'ont plusieurs entités (liées à des contextes d'exécution différents) accédant à un fichier de manière concurrente. En général, un système de fichiers centralisé offre une sémantique de cohérence forte, dite *cohérence séquentielle*. Dans ce cas, chaque mise à jour d'un fichier est immédiatement visible par toutes les entités l'ayant ouvert. Cette sémantique de cohérence, respectée par la plupart des systèmes modernes, est souvent dénommée sémantique POSIX (car elle est imposée par la norme éponyme, qui définit également l'API standard que doit offrir un SGF)[19]. Dans le cas des système répartis, maintenir une cohérence forte pour les différents clients est plus difficile et tous les SGF distribués ne le permettent pas. Il existe ainsi d'autres modèles de cohérence plus lâche, dont notamment :

- la *cohérence temporelle*, qui garantit que les modifications seront rapportées à un client au bout d'un temps borné (quelques secondes) après leur occurrence ;
- la *cohérence par session*, qui garantit que les clients qui ouvrent le fichier après l'occurrence d'une modification la verront.

Implémentation d'un SGF L'organisation sur disque d'un SGF repose principalement sur les notions de *superbloc* et d'*inode*. Le superbloc correspond au premier bloc du SGF et contient toutes les informations cruciales pour son utilisation : identifiant du type de SGF, taille des blocs, nombre total de blocs du volume, lien vers les informations de gestion des blocs libres, lien vers la racine de l'arborescence, etc. Un inode correspond à la structure élémentaire pour stocker les métadonnées associées à un fichier ou un répertoire. Ces caractéristiques schématiques sont communes à tous les SGF mais les détails d'implémentation varient très largement d'un SGF à l'autre. Certaines optimisations ont toutefois été appliquées à de nombreux SGF ; nous en décrivons deux ci-dessous.

[18] ou *File System* (FS) en anglais.

[19] Précisons toutefois que la majorité des SGF ne respectent pas l'intégralité de la norme POSIX. Celle-ci requiert par exemple de maintenir à jour la date de dernier accès à un fichier, action très coûteuse et généralement omise ou effectuée de manière irrégulière.

La plupart des SGF modernes ont recours à une technique (inspirée des bases de données [87]) permettant de mieux protéger leur intégrité contre les effets des pannes. Les systèmes présentant cette caractéristiques sont dit « journalisés » (*journalized/journaling FS*) [52]. Avant de modifier un fichier, les actions à effectuer sont écrites sur disque dans un emplacement spécial : le journal. Si une panne survient avant que le journal soit mis à jour, alors la dernière modification du fichier est perdue. Si la panne survient lors de la modification réelle du fichier, le SGF peut, lors de la reprise, utiliser le journal pour terminer le travail et revenir très rapidement à un état cohérent. Contrairement aux bases de données qui conservent dans le journal l'état complet des données, les SGF n'appliquent généralement le principe de journalisation qu'aux méta-données pour économiser de la place et offrir des performances acceptables. Ce mécanisme est d'une grande utilité pour un compromis relativement modeste (en place disque et en performances d'accès). Néanmoins, il ne constitue pas une garantie absolue contre toutes les possibilités d'altération des informations (qui peuvent survenir à différents niveaux logiciels et matériels, par exemple à cause d'un pilote de disque défaillant), même lorsque les données sont également journalisées.

Les SGF journalisés sont parfois confondus avec les systèmes de fichiers structurés en lot d'enregistrements (*log-structured file systems* ou LFS), qui ne correspondent pourtant pas à la même notion. Puisque le temps de transfert d'un disque est dominé par le placement de la tête de lecture, il convient de favoriser les longs accès séquentiels. Dans ce but, un LFS modifie fréquemment l'emplacement sur disque des informations (données et méta-données), afin de n'effectuer que de longs accès séquentiels au médium de stockage. Le disque est ainsi utilisé de manière similaire à un tampon circulaire. Peu de SGF intègrent aujourd'hui cette technique. Celle-ci est probablement amenée à devenir populaire dans les années qui viennent car le poids de sa motivation initiale s'accroît avec le temps (cf 2.1.1) et sa mise en œuvre est facilité par l'augmentation de la tailles des disques.

2.1.2.4 Système de gestion de bases de données

Un système de gestion de base de données (SGBD) permet de stocker et d'organiser des données de manière plus structurée qu'un SGF en instaurant des liens sémantiques explicites entre les différentes données enregistrées, ce qui augmente notamment l'expressivité de l'interface d'administration et d'interrogation de la base. Un SGBD fournit généralement d'autres propriétés intéressantes dont des fonctions transactionnelles qui permettent d'assurer l'intégrité d'un ensemble de modifications des données. Certaines implémentations de SGBD dépendent d'une interface « fichiers » sous-jacente alors que d'autres sont basées sur une interface de niveau « bloc » et gèrent elles-même tous les aspects liés à l'allocation d'espace, la synchronisation d'accès concurrents, la mise en cache de données, etc. Notre travail n'étant pas centré sur les SGBD, nous n'entrerons pas davantage dans les détails à ce sujet, hormis en 2.5 où nous mentionnerons brièvement les principales topologies de SGBD en grappe.

2.1.2.5 Interface de stockage à objets

Jusqu'à présent les organes de stockage ont été basés sur des interfaces de niveau bloc et une très grande part de la gestion de ces blocs a été à la charge du système d'exploitation[20] (allocation / libération, placement des données, maintien des méta-données...). Pour simplifier la gestion et permettre un meilleur passage à l'échelle des systèmes de stockage, de nombreux travaux de recherche visent aujourd'hui à déporter autant que possible ces tâches au niveau des disques. Ce transfert d'intelligence et d'autonomie vers les périphériques de stockage, envisageable aujourd'hui à coût raisonnable grâce

[20] ou à la charge d'une application « autonome » telle qu'un SGBD

à l'état courant des technologies d'intégration, nécessite cependant l'utilisation d'une interface plus riche que celles des blocs.

La notion d'*interface de stockage à objets* (OBSD ou OSD pour *Object-Based Storage Devices* en anglais) [83, 146] a été développée dans ce but[21]. Dans ce contexte, un objet est un ensemble d'octets auquel sont associées un ensemble de méthodes d'accès et des attributs décrivant les caractéristiques des données (y compris la politique de sécurité les protégeant). Contrairement aux blocs, les objets d'un même disque peuvent avoir des tailles variables et peuvent stocker l'intégralité d'une structure de données de haut niveau (telle qu'un fichier ou une table d'une base de données). La granularité du stockage est ainsi paramétrable ; dans un cas extrême, un objet peut être associé à l'intégralité d'un SGF ou d'une base de données. C'est au périphérique de stockage que revient la gestion de l'allocation de l'espace sur disque et du placement des données, c'est à dire la gestion de structures de données telles que les *inodes* et les tables d'allocation de blocs maintenues jusqu'à présent par le système d'exploitation[22]. Outre la simplification des tâches du système d'exploitation, le fait d'avoir une connaissance des regroupements logiques au niveau du disque permet une meilleure gestion des E/S, car le périphérique peut appliquer des politiques de préchargement et de gestion de cache bien plus précises (en prenant éventuellement en compte le type des données concernées). Certaines approches vont même plus loin en considérant la possibilité de paramètrer le comportement des disques en fonction des applications visées, voire de déporter une partie du travail applicatif (par exemple des procédures de filtrage pour la fouille de données ou les applications multimédia) au niveau d'un disque intelligent [4, 187] .

2.1.3 Intégration des E/S dans les systèmes d'exploitation

Cette section expose et récapitule les différentes couches d'un système d'exploitation qui sont impliquées dans la gestion des E/S.

Une requête d'E/S sur un fichier, démarrée par une application, est d'abord transmise au système d'exploitation (SE) par le biais d'un appel système. Dans le cas d'une requête d'écriture, les données sont recopiées dans l'espace mémoire du noyau.

La requête commence par traverser la couche du *système de fichier virtuel* (VFS en anglais pour *Virtual File System*) : le but de cette couche est de permettre la cohabitation de plusieurs SGF au sein d'un même système tout en assurant la transparence d'une interface d'E/S unique pour les applications. La couche VFS assure donc la conversion entre une requête générique et les opérations correspondantes, spécifiques au système de fichiers déployé sur le point de montage cible de la requête. Cette couche VFS sert également d'intermédiaire avec d'autres types de ressources (différentes d'un SGF local « usuel »), par exemple un SGF distant (voir section 2.4.2.1) ou des SGF spéciaux[23].

Les traitements opérés par le SGF (appelé par le VFS) peuvent être répartis en deux sous-couches :

- la partie supérieure fournit à l'utilisateur l'interface et les structures de données correspondant à la vue logique orientée « fichiers ».
- la partie inférieure gère le couplage entre ces structures de données et leur allocation sur le périphérique logique de stockage. Elle émet donc des requêtes de niveau bloc. Dans les systèmes de stockage à objets, ce rôle est attribué au périphérique de stockage exportant une interface objet.

[21] Il n'existe pas encore de périphériques matériels implémentant ce type d'interface. Des efforts de normalisation sont cependant en cours [211] depuis plusieurs années. Voir également 2.4.5 à ce sujet.

[22] Si nécessaire, l'utilisateur peut cependant agir sur ces aspects en spécifiant des attributs particuliers. Par exemple, un attribut de « qualité de service » peut décrire les contraintes de débit et de temps d'accès requises pour un objet multimédia.

[23] Un exemple classique de SGF spécial est l'arborescence `/proc` de certains systèmes Unix, qui permet d'obtenir via l'interface d'E/S standard, des informations générées dynamiquement sur l'état courant du système.

Les requêtes de blocs du SGF sont adressées au cache global de blocs du système, souvent dénommé *buffer cache*, dont le but est de limiter les (lents) accès au medium de stockage. Ce cache est commun à tous les périphériques bloc du système[24]. De manière courante, la politique de remplacement est de type LRU (*Least Recently Used*) ou horloge et les écritures sont traitées en mode différé *(write-back)*. Selon les implémentations, la taille du cache global peut être fixe (plusieurs dizaines de mégaoctets) ou ajustée dynamiquement en fonction de la pression mémoire du système [35].

Si une requête de blocs ne peut être satisfaite au niveau du cache global (dans le cas d'une lecture) ou si elle concerne une écriture, elle sera transmise au périphérique bloc cible, qui la répercutera, le cas échéant, à d'éventuels périphériques bloc sous-jacents (cas d'un RAID logiciel ou d'un GVL).

Finalement, le pilote de chacun des périphériques physiques impliqués dans le transfert émettra une ou plusieurs requêtes de niveau contrôleur.

Le chemin du retour, c'est à dire de l'acquittement de la commande, est identique. Dans le cas d'une lecture, les données sont copiées de l'espace mémoire du noyau vers celui de l'application.

Nous venons d'exposer le cas le plus général mais certaines précisions et exceptions méritent d'être mentionnées :

E/S brutes La plupart des systèmes fournissent aux applications une interface d'*E/S brutes* (*raw I/O* en anglais) pour manipuler directement un périphérique à blocs. Dans ce mode d'utilisation, une application[25] se substitue aux services génériques du noyau (en particulier à ceux des SGF) dans l'espoir d'obtenir de meilleures performances. Dans la plupart des utilisations des E/S brutes, les données sont transférées directement entre le(s) périphérique(s) physique(s) et l'espace mémoire de l'application[26] ; les blocs transférés ne sont par conséquent pas conservés dans le cache global.

Accès direct au niveau contrôleur Des applications particulières (gravure de CD-ROM, acquisition d'images) requièrent à la fois une interaction forte avec l'utilisateur et un périphérique particulier (graveur de CD, scanner). La plupart des systèmes d'exploitation fournissent une interface SCSI générique, accessible depuis le mode utilisateur, donnant la possibilité à une application de piloter directement un périphérique. Ce type d'utilisation est cependant restreint aux cas marginaux que nous venons d'évoquer.

Synchronisme Il est à noter que les E/S sont généralement synchrones : un processus est bloqué jusqu'à la terminaison de sa requête. Pour permettre un recouvrement entre E/S et exécution de code, la solution majoritaire consiste a utiliser des applications à parallélisme interne (c'est à dire utilisant plusieurs *threads*) mais ceci n'est pas toujours suffisant pour exploiter au mieux les ressources matérielles (et l'utilisation d'un grand nombre de threads peut en outre poser des problèmes de performances). Pour cette raison, le support d'*E/S asynchrones* (qui permettent en outre d'agréger plusieurs requêtes au sein d'un appel système unique) par les noyaux devient courant [26, 67].

Ramdisks Certaines applications avec de fortes contraintes sur la latence d'accès aux données utilisent un « ramdisk », c'est à dire une émulation en mémoire principale d'un périphérique à blocs[27]. Pour des raisons évidentes, un ramdisk requiert une interaction particulière avec le cache global. Le cache global considère chaque bloc du ramdisk comme toujours présent mais ne garde qu'un lien vers le bloc ; aucune copie des blocs du ramdisk dans le cache n'est nécessaire.

[24] Plusieurs tailles de bloc sont gérées, en général 512 octets, 1 ko, 2 ko, 4 ko et 8 ko.

[25] C'est notamment le cas de nombreux SGBD.

[26] On parle aussi d'E/S directes (*Direct I/O*) dans ce cas là.

[27] La persistance des données n'est dans ce cas pas assurée directement. Ce problème est soit ignoré (cas d'un calcul reproductible qui privilégie avant tout les performances), soit résolu partiellement par des sauvegardes périodiques (points de reprise).

La Figure 2.2 synthétise les principales interactions décrites dans cette introduction.

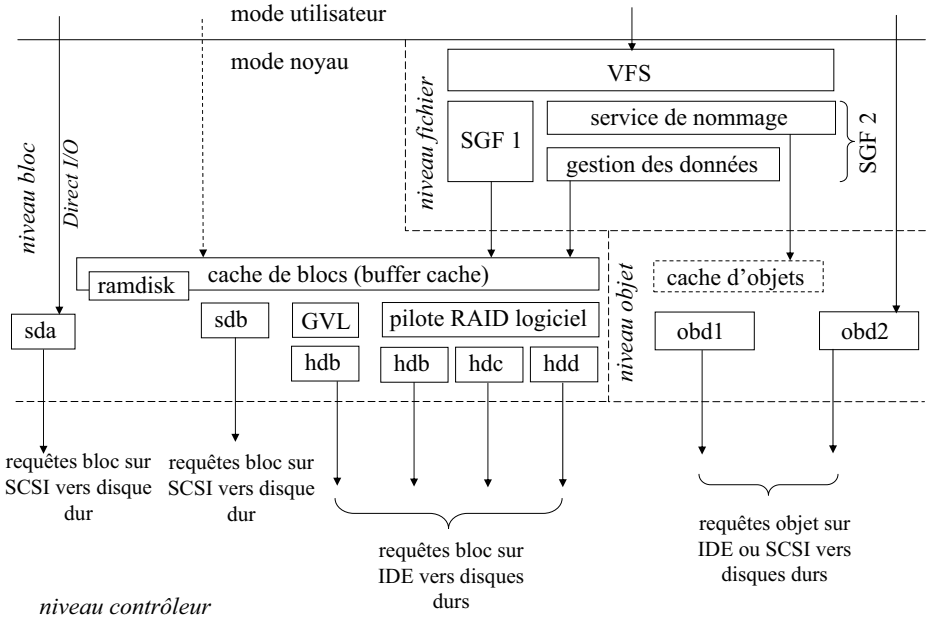


Figure 2.2 – Récapitulatif des principales couches et interfaces utilisées pour la gestion des entrées-sorties dans un système d'exploitation centralisé

Nous venons de décrire les principes de base du stockage de données. Les sections suivantes de ce chapitre décrivent plus précisément, pour chaque niveau d'interface, les techniques employées dans le contexte réparti des grappes de machines. Il est important de noter que des techniques déployées à différents niveaux d'interface ne sont généralement pas exclusives, mais au contraire utilisées de façon conjointe. Ainsi, un *Storage Area Network*, qui permet à plusieurs nœuds de partager des disques au niveau contrôleur, est utilisé en conjonction avec un gestionnaire de volumes logiques (niveau blocs) et un système de fichiers partagé (niveau fichiers).

2.2 Niveau contrôleur (*Storage Area Network*)

Des solutions matérielles ou de très bas niveau logique permettent de multiplexer l'accès à un ou plusieurs disques au niveau contrôleur. Elles sont basées sur l'utilisation d'un réseau spécialisé pour le stockage de données (*Storage Area Network* ou SAN). Un SAN relie les nœuds d'une grappe à un ensemble de disques via une infrastructure de communication, le plus souvent commutée (*switch fabric*) et vise avant tout à partager efficacement des organes de stockage en évitant le passage par un serveur qui constitue un goulot d'étranglement et un point central de défaillance. De plus, si un contrôleur de disques « intelligent » est employé, il est possible de copier/déplacer des données au sein du SAN sans passer par la mémoire du client[28]. Un SAN ne gère cependant pas les problèmes de concurrence et d'accès aux données ; seule une sérialisation des commandes est assurée.

Plusieurs mises en œuvre sont possible pour un SAN, nous décrivons ici les plus courantes.

[28]Cette fonctionnalité est rarement disponible en pratique.

Interconnexion au niveau du bus SCSI Dans ce cas de figure, les cartes SCSI des nœuds sont directement reliées (par des câbles parallèles spécifiques à l'interface SCSI) au contrôleur d'une matrice de disques. Cette configuration est surtout utilisée pour les petites grappes. Les caractéristiques intrinsèques du bus SCSI ne permettent pas un bon passage à l'échelle, que ce soit en termes de nombre de nœuds (15), de performances (le débit et la gestion des commandes concurrentes sont limités), de distance de déploiement (une dizaine de mètres), de souplesse d'administration ou de tolérance aux pannes (un seul chemin sans redondance).

Interface spécialisée de type *Fibre Channel* Des solutions spécifiques ont été développées pour outrepasser ces limitations ; l'une des principales est l'interface Fibre Channel (FC) [115]. FC est une pile de protocoles[29] *médiatrice* : son but est d'assurer le transport à hautes performances de commandes issues d'autres couches protocolaires (notamment SCSI, IP et ATM), plusieurs de ces couches pouvant être supportées simultanément. Au niveau physique, FC fonctionne au dessus de paires torsadées ou de fibres optiques (usage le plus fréquent) et est prévu pour des distances de plusieurs kilomètres. Le matériel courant supporte des débits de l'ordre de 2 à 4 Gbit/s en *full-duplex*. Les topologies point-à-point et boucle arbitrée sont supportées mais la plupart des infrastructures Fibre Channel sont commutées, pour un passage à l'échelle ainsi qu'une tolérance aux pannes accrus.

Dans le contexte du stockage partagé, l'interface Fibre Channel est principalement utilisée pour convoyer efficacement des commandes SCSI entre les nœuds et les contrôleurs des matrices de disques (via le protocole FCP). Le principal problème d'une telle infrastructure est son coût très élevé, même pour les configurations de petite taille. Ceci est dû principalement au recours à des équipements optiques très spécialisés, auquel s'ajoute un important coût humain pour l'administration. De plus, les normes ne sont pas encore uniformément respectées ou interprétées, si bien que de nombreux équipements FC ne sont pas interopérables. Précisons pour finir qu'il est possible, que dans le domaine des interfaces de communication à très haut débit, FC soit détrôné par Infiniband dans un futur proche (ce qui ne résoudra probablement pas les problèmes de coût mentionnés ci-dessus).

Interface IP/Ethernet A partir de la fin des années 90, de nombreux travaux de recherche ont cherché à fournir des solutions SAN plus abordables, plus flexibles, et plus interopérables que celles basées sur des interfaces spécialisées telles que FC. Le projet Netstation [71], cherchait à contourner les problèmes de passage à l'échelle des bus d'E/S en utilisant des périphériques réseau (*Network-Attached Peripherals* ou NAP) ; il fut l'un des premiers à défendre une approche basée sur les protocoles internet « classiques » (TCP/IP et UDP/IP, principalement) [99] avec les arguments suivants :

- Leurs mauvaises performances (par comparaison avec les protocoles spécialisés) ne sont pas intrinsèques (mais plutôt liées aux principales implémentations disponibles) et peuvent être améliorées de façon significative (en augmentant la taille des paquets, en optimisant le calcul des sommes de contrôle, etc.). En outre, si TCP et UDP ne s'avèrent pas concluants, d'autres protocoles plus spécialisés peuvent être définis au dessus de la couche IP.
- Les protocoles internet sont naturellement adaptés pour gérer les communications à longue distance et l'interopérabilité. Ils bénéficient en outre de nombreuses années de recherche et de développement ainsi que d'une très grande base d'utilisateurs (ce qui a contribué à la baisse des prix des équipements, notamment pour le matériel Ethernet).

Un des NAP développés par le projet Netstation fut VISA (*Virtual Internet SCSI Adapter*)

[29] FC-0 à FC-4.

[148] qui émulait un disque directement connecté au réseau, basé sur UDP pour le transport des commandes SCSI.

Depuis, l'idée de bâtir des SAN sur du matériel et des protocoles internet « traditionnels » s'est développée. L'*Internet Engineering Task Force* (IETF) travaille actuellement sur la norme iSCSI (internet SCSI) [149][195]. Une session iSCSI assure le transport de commandes SCSI entre deux entités, un initiateur et une cible, via une ou plusieurs connexions TCP (sur un ou plusieurs liens physiques). Le protocole définit également des fonctionnalités supplémentaires en termes de placement direct de données, de négociation de paramètres de session, de tolérance aux fautes (reprise de session et de connexion, somme de contrôle additionnelle) et utilise les mécanismes existants de la pile IP tels qu'IPSec [117] pour la sécurité (optionnelle). De nombreux fabricants développent et commercialisent actuellement des interfaces iSCSI pour clients (nœuds d'une grappe) et serveurs (matrices de disques directement reliées au réseau ou serveurs exportant leur disques). Cette émergence du stockage « tout IP » est accompagnée par la définition de protocoles d'interconnexion de réseaux SAN Fibre Channel avec IP : FCIP (*Fibre Channel over IP*) [181] gère le transport de commandes FC sur TCP/IP, des passerelles iFCP (*Internet Fibre Channel Protocol*) [151] permettent la migration des éléments d'un SAN de FC vers IP.

Il existe également des approches relativement différentes d'iSCSI pour la mise en œuvre de SAN abordables et interopérables. C'est le cas du protocole HyperSCSI [120], dont la conception est guidée par les points suivants.

- Il existe des différences significatives entre les besoins des protocoles réseau (TCP/IP, UDP/IP) et ceux des protocoles de stockage (SCSI) (au niveau de l'établissement des connexions, du nombre d'utilisateurs, du contrôle de transmission, de l'ordonnancement des messages, etc.). Le coût de traitement lié à la pile TCP/IP est ainsi considéré comme trop important par rapport aux services « inadaptés » qu'elle fournit. Pour ces raisons, HyperSCSI est implémenté directement au niveau Ethernet pour les communications au sein d'un réseau local. Une version basée sur IP est également prévue pour les environnements à grande échelle.
- Les nombreux protocoles basés sur IP utilisés par iSCSI (iSNS pour la découverte de périphériques, IPSec pour la sécurité) sont estimés trop complexes par rapport aux besoins courants, ont une granularité trop limitée et limitent les performances. Il est jugé plus pratique d'intégrer ces aspects « non fonctionnels » au sein du protocole de stockage.

Des tests ont montré des performances significativement supérieures pour HyperSCSI/Ethernet face à iSCSI dans un environnement local (en termes de débit, d'utilisation du CPU et du nombre d'interruptions générées). Ces résultats doivent cependant être tempérés pour l'avenir car il n'y a pour l'instant aucun support industriel d'HyperSCSI. Ce n'est pas le cas d'iSCSI, dont de nombreuses implémentations optimisées risquent d'améliorer sensiblement les performances.

La figure 2.3 résume les principales piles de protocoles pour SAN présentées dans cette section.

2.3 Niveau « blocs »

Au niveau « blocs », on distingue trois principaux types de mécanismes permettant de partager l'accès à un support de stockage permanent.

- La notion de PGB Distant (*Network Block Device* ou NBD) permet à un nœud « client » d'accéder de façon transparente au PGB exporté par un nœud « serveur », par l'intermédiaire d'un réseau (dédié ou non au trafic d'E/S).

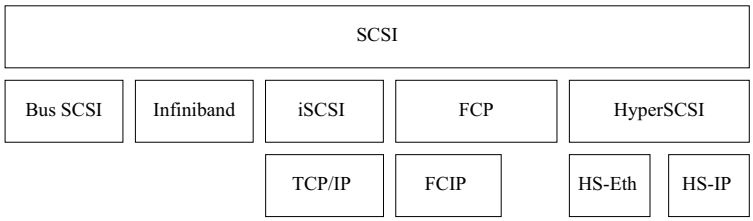


Figure 2.3 – Principales piles de protocoles pour Storage Area Networks

- Le concept de Disque Virtuel Réparti (DVR) étend le principe du RAID à un ensemble de machines mutualisant leurs organes de stockage pour accroître l'espace de stockage, les performances d'E/S et la tolérance aux fautes.
- Un GVL pour grappe (Cluster LVM) permet de définir et d'administrer dynamiquement des volumes logiques partagés entre plusieurs nœuds.

2.3.1 Techniques d'exportation d'un organe de stockage

2.3.1.1 Principes de base

Commençons par décrire le fonctionnement de base d'un NBD, en prenant l'exemple de celui du système Linux [33] - ENBD (*Enhanced Network Block Device*). Au niveau du client, chaque périphérique distant est associé à un PGB (`/dev/nd0`, `/dev/nd1`, etc.) en charge de gérer les communications avec le nœud serveur. Lorsque le nœud serveur reçoit une requête, il la convertit en une requête pour le PGB associé au périphérique (local) concerné. L'acquittement de la requête suit bien évidemment le chemin contraire.

ENBD utilise TCP/IP comme protocole de transport pour garantir la fiabilité et l'ordonnancement des requêtes. Une première implémentation utilisait un thread noyau sur chaque entité (client et serveur). La version courante utilise des threads en mode utilisateur. Plus précisément, au niveau du client, un thread noyau récupère les requêtes soumises au PGB et les transmet à un thread utilisateur en charge d'envoyer les requêtes vers le nœud serveur. Cette méthode est beaucoup moins efficace que la première car elle implique des copies de données supplémentaires (entre l'espace mémoire du noyau et l'espace utilisateur) mais les développeurs d'ENBD justifient cette solution par la souplesse qu'elle procure (notamment pour l'emploi éventuel de canaux de communication sécurisés[30]). Pour limiter le coût des changements de contexte, les requêtes sont transmises par lots (de 10 à 20) au thread utilisateur en charge de l'émission. Plusieurs threads clients (et donc plusieurs canaux de communication) sont en fait utilisés en parallèle pour augmenter le recouvrement entre le traitement des requêtes et les communications sur le réseau. Le parallélisme et la tolérance aux pannes sont bien sûr accrus si les multiples canaux de communication sont couplés à des interfaces physiques différentes. En outre, ENBD gère de façon interne les erreurs récupérables (rupture de connexion, ...), c'est-à-dire sans les remonter aux couches supérieures (RAID logiciel, SGF).

Une limitation majeure d'ENBD tient au fait qu'un périphérique bloc ne peut être exporté que vers un seul client. Son usage est donc restreint à des besoins de sauvegarde (ou de pagination à distance) et, dans le contexte des E/S parallèles, aux topologies où un seul nœud accède à un ensemble de disques (voir 2.3.2.1). D'autres NBD ont été développés pour gérer de multiples clients, notamment GNBD, une couche (inférieure) complémentaire au système de fichiers partagé GFS (voir 2.4.4.2). GNBD utilise des threads noyaux (un par disque distant au niveau du client) et des sockets TCP/IP.

[30] qui utilisent des bibliothèques accessibles uniquement en mode utilisateur.

2.3.1.2 Bénéfices des interfaces spécialisées

Récemment, plusieurs travaux ont cherché à améliorer les performances des NBD en tirant profit des caractéristiques de technologies d'interconnexions spécialisées : fiabilité matérielle des communications, *OS-bypass*, RDMA, processeur embarqué reprogrammable. La performance des transferts est privilégiée par rapport à d'autres critères tels que la sécurité (une grappe est généralement considérée comme un environnement sûr, inaccessible ou relativement isolé de l'extérieur).

GNBD/VIA GNBD/VIA [121] est un exemple d'adaptation simple d'un NBD à une interface de communication optimisée. La pile TCP/IP du noyau est remplacée par une pile protocolaire minimale (avec un interface relativement proche de celle des sockets pour minimiser les modifications du code de GNBD) déléguant la plupart des contraintes de fiabilité et de contrôle de flux à la carte réseau. Les résultats montrent des performances d'accès supérieures à la version initiale de GNBD (sur TCP/IP) et comparables à celles d'un disque local. Aucune quantification ni evaluation de la charge induite (sur le client comme sur le serveur) n'ont cependant été effectuées.

OPIOM Le projet OPIOM (Off-Processor I/O with Myrinet) [80, 79] s'est inscrit dans le contexte des serveurs de vidéo à la demande (*Video on Demand* ou VoD) en grappe. Son objectif principal était la réduction de la charge induite par les E/S sur les nœuds exportant leurs disques[31], en réduisant le cheminement de données entre le disque distant et l'application d'où émanent les E/S.

Commençons par illustrer le fonctionnement d'une application « naïve » utilisant les capacités d'*OS-bypass* d'un réseau pour transférer des blocs de données.

Pour une requête de lecture d'un nœud client (C) vers un nœud serveur (S), le cheminement classique des données peut être résumé par les étapes décrites ci-dessous et représentées sur la figure 2.4 (a).

1) Sur C, l'application cliente émet une requête vers S (grâce à l'*OS-bypass*, la requête ne traverse pas les couches noyau de C, ni de S.)

2) Sur S, l'application serveur reçoit la requête et envoie une requête d'E/S au disque local.

3) Sur S, le disque récupère les données et les transfère au contrôleur SCSI (via le bus SCSI), qui les transfère à son tour en mémoire principale (dans l'espace du noyau) par DMA via le bus d'E/S (PCI).

4) A la réception de l'interruption (d'acquittement) émise par le contrôleur SCSI, le noyau met à jour le buffer cache, et copie les données dans l'espace de l'application. Cette copie nécessite l'intervention du processeur et l'utilisation du bus mémoire.

5) Enfin, les données sont envoyées au nœud client C en utilisant le protocole de communication du réseau spécialisé, sans repasser par le noyau de S. Cette opération nécessite toutefois le transfert de données (via le bus mémoire et le bus PCI) dans la mémoire de la carte réseau.

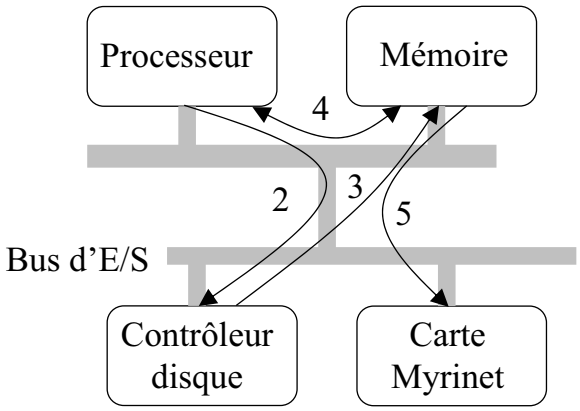
L'approche d'OPIOM vise à limiter le cheminement des données, en particulier sur le bus mémoire et le bus d'E/S qui sont les goulots d'étranglement [16]. Elle repose sur le fait que certains périphériques PCI (dont les contrôleurs SCSI et certaines cartes réseau) sont équipées de moteurs de DMA, d'assez de mémoire couplée dans l'espace d'adressage PCI, et capables d'effectuer des transferts de ou vers la mémoire d'autres périphériques PCI. Ainsi, le contrôleur du disque peut directement transférer les données lues dans la mémoire de la carte réseau (cf figure 2.4(b)).

Le prototype a été implémenté sur une grappe de Pentium II sous Linux avec un réseau Myrinet. Il est constitué d'une bibliothèque liée à l'application serveur et d'un module noyau pour rediriger les

[31] Le travail s'est concentré sur les accès en lecture, largement majoritaires dans le cadre de la VoD.

(a) Serveur sans OPIOM :

1) réception de la requête par le serveur



(b) Serveur avec OPIOM :

1) réception de la requête par le serveur

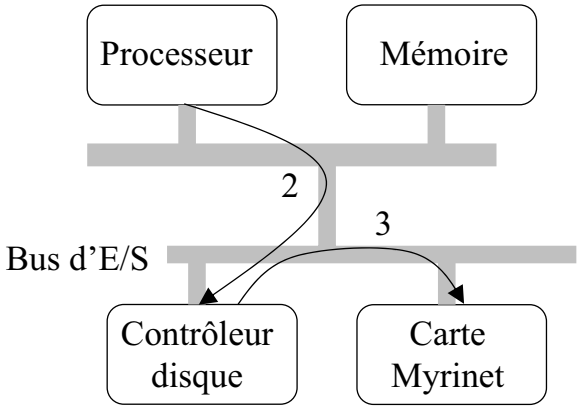


Figure 2.4 – Lecture de données sur un serveur via Myrinet

données lues vers la carte réseau du serveur. Ce concept a été développé dans un contexte applicatif (serveur de vidéo à la demande) mais il peut être adapté aisément à celui d'un PGB (dans ce cas, la capacité d'OS-Bypass et la bibliothèque applicative ne sont plus nécessaires).

Des tests ont montré des gains substantiels par rapport à une l'utilisation « naïve » de l'OS-Bypass décrite précédemment en termes de débit, de latence, de gigue et d'utilisation du CPU du serveur.

READ^2 On peut décrire $READ^2$ [55, 56] comme la continuité des travaux d'OPIOM dans le cadre du calcul *Out of Core*[32], application très sensible aux performances de la pagination et donc à celles des E/S ainsi qu'à l'encombrement du bus mémoire. L'optimisation supplémentaire apportée par $READ^2$ par rapport à OPIOM est le déplacement du pilote du disque serveur. Dans le cas d'OPIOM (et dans les autres approches précédemment évoquées dans cette section), c'est au CPU du serveur que revient la gestion des requêtes d'E/S (vers le disque local). Dans celui de $READ^2$, la gestion des requêtes clientes est implémentée au niveau du processeur de la carte réseau. Ainsi, il n'y a aucune utilisation du processeur du serveur et les accès concurrents (émanant de différents clients) sont traités au niveau de la carte réseau. Cette approche comporte cependant son lot d'inconvénients : elle nécessite du matériel très spécialisé (carte réseau avec processeur embarqué dont le microcode est reprogrammable) et du code très dépendant du matériel (technologie d'interconnexion et famille de contrôleur disque[33]). $READ^2$ fournit une bibliothèque applicative pour faire des E/S directes (qui nécessite une modification du code applicatif) ainsi qu'un PGB pour l'utilisation d'un système de fichiers. Dans ce second cas d'utilisation (PGB), un problème important est éludé : lorsqu'une écriture est effectuée par un nœud A sur un disque hébergé par un nœud B, le cache global de B peut être désynchronisé par rapport aux données sur le disque car l'écriture transite directement de la carte réseau vers le disque, sans passer par le noyau[34]. Un mécanisme d'invalidation du cache est donc nécessaire. Ainsi, si l'on souhaite qu'un nœud exportant ses disques en écriture puisse également y accéder de manière cohérente[35], il est nécessaire d'interrompre le nœud serveur pour invalider son cache. Le seul contournement possible est de désactiver le cache global de B, ce qui est très pénalisant en termes de performances et va à l'encontre de l'objectif initial.

$READ^2$ peut également être vu comme un prolongement des travaux de DREAD (Direct Remote Access to Devices) [65] qui permet à un nœud de piloter à distance le contrôleur SCSI d'un nœud serveur par l'intermédiaire d'un réseau SCI. DREAD ne supporte qu'un seul client et correspond à une approche de plus bas niveau : le client utilise une interface contrôleur et non bloc.

Le prototype $READ^2$ a été implanté sur une grappe de 8 nœuds bi-Athlon sous Linux avec un réseau Myrinet et des contrôleurs SCSI compatibles LSI53c8xx. Des tests ont comparé les performances d'une application de calcul (intensive au niveau du processeur) en grappe avec deux configurations pour les E/S : GFS[36]/GNBD/TCP/IP/GM et GFS/$READ^2$. Les accès disques obtiennent des performances comparables dans les deux cas mais l'apport de $READ^2$ pour le décharger le CPU des nœuds serveurs est sensible (gain de 17% en temps d'exécution).

2.3.2 Disque virtuel Réparti / RAID sur grappe

Nous distinguons les architectures de Disque Virtuel Réparti (DVR) selon qu'elles acceptent un client unique ou plusieurs clients. Dans le premier cas, il s'agit simplement d'une extension assez simple des principes RAID développés dans un contexte centralisé. Dans le second cas, il peut y avoir nécessité de synchronisation entre les clients pour les accès aux données ainsi que la gestion des défaillances.

Un exemple classique de tels besoins de synchronisation est l'utilisation répartie des schémas RAID basés sur des informations de parité (RAID 5 étant le plus utilisé d'entre eux). Le problème

[32] Le calcul *Out of Core* désigne les applications de calcul scientifique travaillant sur un volume de données largement supérieur à la taille de la mémoire principale des machines utilisées. Voir par exemple [41] pour plus de détails sur ce sujet.

[33] N.B. : Même pour une interface donnée, par exemple SCSI, il existe plusieurs familles de contrôleurs, dont la programmation est différente.

[34] Le même problème se pose pour OPIOM, mais cet aspect n'a pas été traité car le prototype n'a été développé que pour les accès en lecture.

[35] Ce qui est généralement le cas ; les disques sont souvent exportés de manière symétrique.

[36] Voir section 2.4.4.2 pour une présentation du système de fichiers partagé GFS.

vient du fait qu'il n'y a plus de point central de coordination[37]. Des requêtes associés à des blocs distincts peuvent entrer en concurrence si ces blocs font partie de la même bande de données. L'entrelacement de telles requêtes peut conduire à l'incohérence des informations de parité, et, par extension, à la perte des données lors d'une panne[38]. Il est important de noter que ce problème est lié à la mise en œuvre d'un PGB réparti et qu'il peut se produire même si les couches supérieures du système utilisent des mécanismes de synchronisation (ceci est lié au fait qu'une distribution de données à la RAID 5 impose implicitement des relations entre les différents blocs qu'elle gère). Des protocoles de synchronisation ont été définis pour résoudre ce problème [7], mais leur complexité et le coût qu'ils imposent sont tels que les schémas de redondance par parité n'ont généralement pas été utilisés, jusqu'à présent, en l'absence d'un contrôleur central[39].

2.3.2.1 Architectures pour client unique

On peut distinguer deux types d'architectures pour client unique : d'une part, une extension directe du RAID centralisé et, d'autre part, une gestion explicite de la distribution des données.

RAID Distribué L'exemple le plus simple d'un telle architecture correspond à l'emploi de plusieurs instances d'un NBD (ENBD, par exemple) combiné à une couche RAID logicielle. Ce cas est très proche d'une architecture centralisée. En effet, il n'y a pas de problème supplémentaire de cohérence des données à gérer (quel que soit le niveau RAID employé) et la panne d'un disque (ou d'un nœud distant hébergeant un disque) ainsi que l'éventuelle phase de reconstruction sont identiques.

Approche Maître/Esclave Certaines infrastructures pour client unique gèrent cependant la distribution de manière explicite. C'est notamment le cas de DRBD (Distributed Replicated Block Device) [185], qui vise à fournir un support de stockage maître-esclave pour services dupliqués. Un service S (par exemple une base de données) s'exécute sur un nœud A « primaire » et utilise un PGB particulier, qui lit et écrit les données localement et qui les transfère également sur le PGB d'un nœud B distant (« secondaire »). En cas de panne de A, celle-ci est est détectée par un mécanisme d'appartenance (*(cluster) membership service*, basé sur un principe de *heartbeat*[40]), le service S est alors démarré sur B qui devient le nœud primaire et prend le contrôle du PGB dupliqué. Un nouveau nœud secondaire peut alors être ajouté pour tolérer une éventuelle défaillance de B. DRBD gère ainsi la synchronisation des données et les différents scénarios de reprise sur panne. Dans le cas d'un fonctionnement normal, trois modes de synchronisation sont possibles, chacun correspondant à un compromis entre performance des E/S et fiabilité des données.

Mode 1 Les écritures ne sont pas synchronisées. Sur le nœud primaire, le PGB DRBD acquitte une requête dès qu'elle a été effectuée sur le disque local.

Mode 2 La requête n'est acquittée qu'après avoir reçu une confirmation du nœud secondaire indiquant que la requête a bien été reçue (mais pas encore écrite sur le disque secondaire).

Mode 3 La requête n'est acquittée que lorsque l'écriture sur le disque du nœud secondaire a eu lieu.

Les modes 1 et 2 sont bien sûr plus efficaces mais offrent moins de garanties de tolérance aux pannes. En particulier, l'ordonnanceur de disque du nœud secondaire peut réorganiser des requêtes d'écritures

[37] Ce rôle est endossé par le contrôleur RAID dans un cadre centralisé.

[38] Voir par exemple [7] pour plus de précisions sur le sujet.

[39] Cette approche décentralisée a toutefois un meilleur potentiel de passage à l'échelle et devrait gagner de l'intérêt progressivement, alors que le nombre de clients supportés par les systèmes de stockage augmente

[40] Un mécanisme d'appartenance permet la détection de pannes de type *fail-stop*. Il est basé sur le fait que les nœuds communiquent périodiquement entre eux pour attester de leur bon fonctionnement.

(afin d'optimiser les transferts). Ceci peut être très problématique au niveau de la cohérence de méta-données critiques (du SGF ou de la base de données déployé(e) au dessus de DRBD) telles que le journal des opérations (qui permettent justement de maintenir/rétablir un état cohérent des données). Pour remédier à ce problème, DRBD infère les dépendances entre écritures successives et emploie une barrière de synchronisation lorsque cela est nécessaire.

Le prototype DRDB ne gère qu'un seul esclave. En cas de pannes de l'un des nœuds, un nouveau nœud secondaire peut être ajouté et resynchronisé mais ceci nécessite l'intervention d'un administrateur humain.

2.3.2.2 Architectures pour clients multiples

Avec une telle architecture, plusieurs nœuds d'une grappe peuvent mettre en commun leurs disques locaux.

RVSD Un des premiers travaux dans le domaine fut RVSD (*Recoverable Virtual Shared Disk*) [18], développé par IBM pour le système AIX et principalement destiné aux applications de type bases de données, utilisant des E/S directes. RVSD peut également être utilisé conjointement avec le cache global du système mais aucune gestion de la cohérence des caches distribués n'est assurée ; ce type d'emploi n'est donc possible que pour des données en lecture seule ou exportées vers un client unique.

L'architecture est basée sur un NBD en mode noyau, implémenté uniquement par interruptions au niveau du serveur (c'est à dire sans recourir à un ou plusieurs processus spécifiques, pour limiter le coût de la gestion des E/S) et utilisant une couche de datagrammes bâtie au dessus d'IP[41]. Un cache optionnel[42] (*write-through*) est implémenté au niveau serveur, pour l'ensemble des PGB exportés par un nœud.

Le NBD est utilisé pour permettre le partage de disques entre plusieurs nœuds. Plus précisément, un nœud donné peut agir comme serveur de certains PGB et client pour d'autres. Il peut bien sûr accéder à ses PGB locaux sans passer par l'infrastructure de communication de RVSD (une synchronisation avec le cache serveur est toutefois nécessaire). Un gestionnaire de volumes logiques est utilisé pour fournir une vue simplifiée de l'espace de stockage aux applications.

Enfin, RVSD gère de façon transparente les pannes de serveurs. Deux pré-requis sont nécessaires : chacun des disques doit être physiquement accessible par deux nœuds (par exemple via l'emploi de câbles SCSI) et un mécanisme d'appartenance à la grappe permet de détecter les nœuds défaillants. Lorsqu'un nœud serveur tombe en panne[43] (ou n'est plus accessible par l'ensemble des clients), un protocole à deux phases permet d'abord l'invalidation du serveur auprès de tous les clients (et également l'arrêt du serveur, si celui-ci est encore en vie) puis le transfert du rôle de serveur vers le second nœud relié au disque.

Petal Le projet Petal [133, 134] de Digital Equipment correspond à l'une des premières implémentations de RAID distribué. L'objectif du projet est la conception d'un système de stockage passant à l'échelle (en nombre de clients, de requêtes et en volume de données stockées), pouvant être administré dynamiquement et tolérant les pannes. Un système Petal est constitué d'un ensemble de serveurs

[41] Aucune précision n'est fournie pour expliquer le fonctionnement précis de cette couche et en quoi elle diffère du protocole UDP. Seule la portabilité d'IP est mentionnée.

[42] Ce cache n'a d'intérêt que dans le cas des E/S directes ; dans le cas contraire (E/S brutes), il y a redondance avec le cache global. Il ne pose pas les mêmes problèmes de cohérence que le cache global car il est implémenté au niveau de chaque serveur, alors que c'est la synchronisation des caches clients qui est en cause.

[43] Les pannes considérées sont principalement de type *fail-stop* mais le mécanisme gère également les problèmes tels que les partitions de réseau qui entraînent une vue incohérente de l'état des différents nœuds.

de stockage coopératifs (aucun service n'est centralisé). L'état global du système (liste des membres et des volumes présentés aux clients) entre les différents serveurs est maintenu par l'emploi d'un algorithme de consensus distribué (Paxos[44] [130]).

Le concept fondamental de Petal est celui de « disque virtuel » ou « volume virtuel » (VV). Un VV abstrait un ensemble de disques répartis sur plusieurs nœuds. De manière identique au concept de mémoire virtuelle, un VV définit une plage d'adresses et couple, au moment où cela est nécessaire, une plage d'adresses à une zone de stockage physique. Ce principe s'avère très avantageux en termes de souplesse et de reconfiguration dynamique : la taille de l'espace (physique) de stockage ainsi que le nombre de serveurs peuvent évoluer, le schéma de répartition des données (et donc le niveau de redondance) peut être modifié, un *snapshot*[45] efficace peut être facilement implémenté en utilisant un mécanisme de *copy-on-write*. Une table de correspondance à trois niveaux, dont les deux premiers sont dupliqués et synchronisés sur chaque serveur, permet de gérer les couplages. Elle est organisée de la façon suivante :

- Le premier niveau traduit l'identifiant de disque virtuel (spécifié par un client) en un identifiant de cartographie globale du disque (GMap).
- A partir de la GMap et du décalage (*offset*) fourni par le client, le deuxième niveau détermine quel serveur est responsable de la traduction de l'offset.
- Enfin, sur le nœud indiqué par la phase précédente, le dernier niveau permet d'obtenir le disque physique et la position sur celui-ci qui correspondent au doublet (GMap, offset) recherché.

Pour optimiser les performances d'E/S et obtenir un système de stockage tolérant aux pannes (de nœud, disque ou réseau), les données écrites par les clients sont étalées (*striping*) sur plusieurs disques. Comme mentionné précédemment, plusieurs schémas de répartition peuvent être utilisés par le système ; les travaux sur le prototype Petal se sont focalisés sur la topologie de *chained-declustering* [101]. Son principe est assez proche des niveaux de RAID vus en section 2.1.2.2 ; la principale différence est qu'en cas de panne d'une entité (nœud, disque), la charge est mieux répartie sur les entités « survivantes ». Ce schéma ne supporte que la panne d'un seul disque et s'avère, en outre, moins robuste qu'un schéma de type RAID 1+0.

Les clients utilisent un protocole particulier (basé sur des appels de procédures à distance) pour dialoguer avec les serveurs. Aucune information d'état sur les disques virtuels n'est maintenue au niveau du client, hormis une liste des serveurs déjà connus et des heuristiques permettant de trouver rapidement le serveur approprié. Lorsqu'une requête est transmise à un serveur inadéquat, celui-ci retourne une erreur et indique au client le gestionnaire courant des blocs spécifiés.

Le prototype Petal a été implémenté sous Digital Unix en mode utilisateur, avec des accès aux PGB physiques sous forme d'E/S brutes. Les serveurs communiquent entre eux par l'intermédiaire de datagrammes UDP. Les machines, clients comme serveurs, sont interconnectés par un réseau ATM commuté. Le schéma de redondance ne permet la survie des données qu'à la panne d'un seul composant (nœud, disque, ou lien réseau vers un nœud).

Enfin, un système de fichiers distribué (Frangipani) a été conçu de façon complémentaire à Petal (voir 2.4.4.4).

[44] Cet algorithme garantit la cohérence du système malgré un nombre arbitraire de pannes de nœuds et du réseau (suivies d'éventuelles reprises) ainsi que sa réactivité, dés lors que la majorité des nœuds peut continuer à communiquer.

[45] Un mécanisme de *snapshot* permet de conserver une image figée et cohérente de l'état courant d'un volume logique. Les images sont gérées de façon incrémentale, plusieurs versions d'une même image peuvent partager les mêmes blocs physiques si leur contenu est identique.

Tertiary Disk Le projet Tertiary Disk[46] [218, 17], initié à l'université de Berkeley, a pour ambition de fournir un système de stockage de masse (pour des applications telles que les bases de données multimédia ou l'archivage du Web) efficace, extensible et relativement bon marché. Cette approche est présentée comme un contournement des limitations inhérentes aux matrices de disques, à savoir un mauvais passage à l'échelle des performances et du rapport coût/capacité. La démarche de Tertiary Disk par rapport aux matrices de disques est assez analogue à celle des grappes de stations par rapport aux machines massivement parallèles (MPP).

Contrairement à des infrastructures logicielles telles que Petal, qui n'ont aucun pré-requis particulier de niveau matériel, Tertiary Disk est bâti sur une architecture relativement atypique, bien qu'à base de composants standards. Un nœud logique est défini comme deux PC partageant un ensemble de disques (via un ou plusieurs bus SCSI communs). Ce patron de conception est motivé par des raisons de tolérance aux pannes. Dans le cadre d'un fonctionnement normal, chaque PC est responsable de la moitié des disques. Dans le cas d'une panne[47], le PC survivant prend en charge la totalité des disques auxquels il est connecté. Plus généralement, le système n'a aucun point central de défaillance.

Au niveau logiciel, l'exploitation de l'architecture Tertiary Disk est basée sur le système de fichiers réparti xFS[48] (décrit en 2.4.3.3), qui ne nécessite aucun serveur centralisé. Chaque fichier est associé à un « groupe de bandes » (*stripe group* ou SG). Chaque SG est associé à un ensemble de disques (un disque par nœud logique) ainsi qu'à un schéma de redondance (RAID 5 le plus souvent ; le calcul de parité est géré au niveau du client pour limiter la complexité du serveur et les synchronisations entre serveurs).

Le prototype développé utilise dix nœuds logiques interconnectés par des réseaux Myrinet et Fast Ethernet. Chaque nœud héberge quatre contrôleurs SCSI ainsi qu'une quarantaine de disques. Le système global stocke plus de 3 To de données.

RAID-X Le projet RAID-X [105, 106], développé conjointement à l'université de Hong Kong et à l'University of Southern California a apporté deux contributions principales à la thématique des disques virtuels répartis : une nouvelle architecture logicielle ainsi qu'un schéma de distribution des données[49].

L'architecture du système RAID-X est fondée sur le principe d'espace unique d'E/S (Single I/O Space ou SIOS) [104], qui permet à chaque nœud de la grappe d'addresser de manière uniforme tous les blocs des différents organes de stockage, qu'ils soient locaux ou distants. Cette idée a déjà été abordée, notamment par Petal au niveau utilisateur et par xFS au niveau du système de fichiers. Dans le cas de RAID-X, cette abstraction est implémentée au niveau du noyau, ce qui induit plusieurs avantages : d'une part, des performances accrues car moins de changements de contexte et de copie de données sont nécessaires, d'autre part, une plus grande transparence (il n'est pas nécessaire de modifier les applications ni les couches hautes du système et différents systèmes de fichiers peuvent être utilisés). En outre, la présence d'un SIOS au niveau du noyau peut simplifier significativement l'implémentation d'autres services pour système à image unique tels qu'une mémoire virtuelle partagée distribuée, un système global de points de reprise ou un espace de processus unique permettant la migration de tâches.

Chaque nœud d'un système RAID-X utilise un module noyau nommé CDD (*Cooperative Disk Driver*), lui-même divisé en trois sous modules : client, gestionnaire de disque local et gestionnaire de cohérence des données. Les CDD coopèrent de manière pair-à-pair (via des connexions TCP/IP) :

[46] Le nom du projet reflète son double objectif : obtenir un coût par Mo et une capacité digne des bandes magnétiques tout en bénéficiant de la performance des disques.

[47] Les pannes considérées sont les pannes de machine, de disque, de contrôleur SCSI et de câble SCSI.

[48] Contrairement a Frangipani qui a été développé au dessus de Petal, xFS a été adapté à l'architecture Tertiary Disk.

[49] Le nom du projet correspond au nouveau niveau RAID ainsi défini.

aucun serveur centralisé n'est nécessaire, et chaque CDD peut être à la fois client et serveur. Le schéma de distribution est implémenté au niveau du client et peut être modifié. Le gestionnaire de cohérence prend en charge les problèmes de synchronisation des clients (caches[50], mise à jour des informations de parité dans le cas du RAID 5 réparti, etc.) grâce à un système de verrous distribués (lecteurs multiples / écrivain unique), attribués et relâchés de manière atomique. Contrairement à Petal, aucun aspect de gestion dynamique (outre la tolérance aux pannes fournie par la redondance des données), tel que l'ajout de nœuds, de disques, la migration de données ou la modification du schéma n'est pris en compte.

L'autre axe majeur du projet est projet est la définition d'un nouveau schéma de répartition des données nommé *Orthogonal Striping and Mirroring* (OSM). L'objectif d'OSM est d'optimiser les performances en écriture (qui sont un problème notoire du RAID 5, notamment pour les données de petite taille) ; un système de sauvegarde de points de reprise optimisé pour OSM a d'ailleurs été proposé [105]. Ce schéma ne tolère qu'une seule panne de disque ou de machine, à l'instar de RAID 5. Il est cependant plus coûteux (50% de données redondantes) et ne peut tolérer certains cas de pannes multiples, contrairement au RAID 1+0 et au *Chained Declustering*, qui ont le même taux de redondance.

Un prototype a été développé sur une grappe de 16 PC sous Linux, interconnectés par un réseau Fast Ethernet. Les expériences ont été focalisées sur la démonstration des meilleures performances d'OSM (par rapport à RAID 1 et RAID 5) dans le cas d'un fonctionnement normal (sans pannes).

DRAID DRAID [62] est un projet de l'université de Gênes. Sa motivation principale est issue du constat qu'aucun des systèmes précédemment décrits dans cette section n'est capable de tolérer plus d'une panne de disque (du moins au niveau d'un même *stripe group*) ; cet état de fait a été jugé rédhibitoire pour l'utilisation de disques virtuels répartis sur des parcs matériels (machines et disques) de grande taille.

Pour répondre à ce besoin, un nouveau schéma de distribution de données a été conçu. Celui-ci est fondé sur les codes d'erreurs de Reed-Solomon ; il permet de paramétrer le niveau de tolérance aux pannes tout en limitant le surcoût de la redondance (par rapport à une approche RAID 10 supportant le même nombre de pannes). Un bloc logique (de taille T) associé à un PGB DRAID est découpé en N segments. A ceux-ci, s'ajoutent K segments correcteurs d'erreurs (K étant paramétrable en fonction du degré de tolérance aux pannes souhaité). Une bande de données correspond ainsi à (N+K) segments de taille (T/N). Puisqu'il est peu imaginable que le degré de parallélisme du système soit infini, une matrice DRAID avec de nombreux nœuds comporte plusieurs lignes de (N+K) nœuds.

L'architecture du système DRAID est assez proche de celle de RAID-X : elle implémente également l'abstraction de SIOS. Sur chaque nœud, un module DRAID utilise un ensemble de threads pour trois types d'opérations : l'émission de requêtes vers d'autres nœuds, la reception de requêtes et la gestion des E/S locales. Les communications utilisent UDP/IP. Le système est peu administrable ; la seule éventualité prévue concerne l'ajout de nœuds (N+K nœuds peuvent être ajoutés pour former une nouvelle ligne de la matrice de disques). Toutefois, les auteurs ne précisent pas comment un tel ajout peut être répercuté de façon synchronisée au niveau des différents modules DRAID coopératifs, ni au niveau des services de plus haut niveau utilisant le disque virtuel réparti (un SGF, par exemple).

Lorsqu'un problème d'écriture d'un segment est détecté, ce segment est temporairement relogé sur une autre nœud (*backup station*, sur la même colonne mais sur un nœud différent de la matrice). Si la panne est transitoire, il peut se produire un décalage entre la version du segment stockée sur le nœud défaillant et celle de la station de reprise. Pour détecter ce problème, tous les segments sont

[50]Le gestionnaire de cohérence collabore avec le cache global de blocs (*buffer cache*) du système et invalide les blocs lorsque cela est nécessaire. RAID-X est l'un des rares systèmes de niveau blocs à gérer la cohérence des caches. La plupart des infrastructures la prennent en compte au niveau fichiers.

accompagnés d'un numéro de version et les numéros de versions de tous les segments composant une bande sont vérifiés à chaque lecture. La même méthode est employée pour résoudre les problèmes d'écritures concurrentes sur un même bloc logique du PGB. Après une écriture, une lecture vérifie que tous les segments d'une bande sont bien associés à la même version. Si ce n'est pas le cas, chaque client incrémente son compteur de version d'une valeur aléatoire et réessaye jusqu'à l'obtention d'un état cohérent[51].

Le prototype DRAID a été implémenté sur une grappe de 8 PC sous Linux, interconnectés par un réseau Gigabit Ethernet. Seule une configuration de type « 6+2 » a été testée. Les expériences ont été restreintes à la mesure de grands accès séquentiels effectués par un client unique. Dans cette configuration, les performances en écriture sont satisfaisantes (le réseau est saturé lorsque plusieurs requêtes sont agrégées) mais les lectures sont limitées par la congestion imposée sur le commutateur, UDP n'effectuant aucun contrôle de flux.

Le tableau 2.3 récapitule les principales caractéristiques des disques virtuels présentés ci-dessus.

[51] Cette solution paraît peu satisfaisante. Utiliser des numéros de version incluant un identificateur du client permettrait d'éviter l'incrémentation aléatoire qui ne garantit pas la détection d'incohérences. En outre, cette méthode implique une lecture supplémentaire de toute une bande pour chaque écriture ; ceci peut s'avérer très pénalisant si une bande de données contient beaucoup de segments, ce qui risque d'être le cas si l'on souhaite disposer d'un système tolérant de nombreuses pannes.

Caractéristiques	**RVSD**	**Petal**	**Tertiary Disk**	**RAID-X**	**DRAID**
Pré-requis matériels (pour la tolérance aux pannes)	Chaque disque doit être accessible par 2 nœuds	Aucun	Chaque disque doit être accessible par 2 nœuds	Aucun	Aucun
Schéma de répartition des données modifiable ? (statiquement / dynamiquement)	Non / Non	Oui / Oui	Oui / Non	Oui / Non	Non / Non
Schéma(s) de répartition des données implémenté(s)	Non géré (délégué à une couche logicielle de plus haut niveau)	Striping et *Chained Declustering*	RAID-5	RAID-5, RAID-10, Chained Declustering et *OSM*	Reed-Solomon (avec redondance paramétrable)
Mécanisme de SIOS	Informations de configuration (statiques) identiques sur tous les nœuds	Pilotes coopératifs au niveau utilisateur	Non géré (délégué aux serveurs du système de fichiers distribué xFS)	Pilotes coopératifs au niveau noyau	Pilotes coopératifs au niveau noyau
Mécanisme de maintien de la cohérence des données	Non géré	Algorithme Paxos pour synchroniser les métadonnées	Non géré (délégué au SGF xFS)	Verrous au niveau des pilotes	Numéros de version associés aux segments de données
Maintien de la cohérence des caches clients ? (buffer cache)	Non	Non	Non	Oui	Non
Mécanisme de tolérance aux pannes	Mécanisme d'appartenance + redondance physique des accès aux disques + protocole de reprise à deux phases (un seul nœud de reprise, défini statiquement, pour chaque serveur)	Mécanisme d'appartenance + Paxos + module d'accès aux données au niveau du client (dépendant du schéma de distribution)	Redondance physique des accès aux disques	Uniquement celle fournie par la redondance des données	Mécanisme de relogement d'un segment (sur un nœud déterminé statiquement) lorsqu'un problème écriture est détecté

TAB. 2.3 – Caractéristiques des principaux systèmes de disques virtuels distribués

2.3.3 Gestionnaires de volumes logiques pour grappes

L'intérêt d'un gestionnaire de volumes logiques a déjà été abordé en 2.1.2.2. L'objectif de cette section est d'expliciter les fonctionnalités attendues d'une telle couche dans le contexte réparti des grappes. Les principales missions d'un gestionnaire de volumes logiques pour grappe (GVLG ou CLVM pour *Cluster Logical Volume Manager*) sont :

- **Abstraire les ressources de stockage** (disques, partitions). Par rapport à un cadre centralisé, la contrainte supplémentaire du GVL est de maintenir une vue cohérente des volumes (et l'intégrité de ces derniers) auprès de tous les nœuds concernés, qui exécutent chacun une instance du système d'exploitation.
- **Permettre la migration de données et les sauvegardes sans interruption de service**. De nombreuses applications critiques requièrent une continuité d'accès aux données alors que leur emplacement physique est en cours de modification ou qu'une sauvegarde est effectuée.
- **Permettre une administration efficace et potentiellement répartie des volumes**. Pour simplifier la tâche des administrateurs, les opérations de maintenance sur les différents volumes accessibles au sein d'une grappe doivent pouvoir être effectuées depuis un point central. Cependant, pour des raisons évidentes de tolérance aux pannes, le GVLG doit pouvoir être administré depuis n'importe quel nœud.
- **Contrôler l'accès aux volumes**. Le GVLG doit être en mesure de contrôler les accès aux différents volumes (le niveau de granularité est généralement le nœud), pour éviter la perte de données. Ce type d'incident peut être du à une intrusion malveillante ou à une erreur de configuration, mais il peut également être causé par la panne byzantine (matérielle ou logicielle) d'un nœud.

Une conséquence importante de ces missions (et notamment des deux dernières), est que chaque instance du GVLG doit disposer d'une vue cohérente des nœuds en état de marche. A cet effet, il doit interagir avec un gestionnaire de grappe (*Cluster Manager*), basé sur un mécanisme d'appartenance, qui permet de détecter les nœuds en panne et d'assurer l'intégrité du système en grappe malgré une éventuelle partition du réseau (à l'aide, respectivement, de mécanismes de *heartbeat* et de *quorum*). Pour éviter l'altération de données due à la panne (potentiellement byzantine) d'un nœud, le GVLG s'appuie sur un mécanisme de « clôture des E/S » (*I/O fencing*). Lorsque, sur nœud, le gestionnaire de grappe détecte la panne d'un autre nœud, il avertit le GVLG qui prendra la mesure adéquate pour empêcher tout futur accès aux volumes effectué par le nœud défaillant, jusqu'à l'intervention d'un administrateur qui permettra de le réintégrer au groupe des nœuds actifs. Plusieurs méthodes d'*I/O fencing* sont envisageables [175].

- Si le matériel le permet (au niveau d'un commutateur administrable ou du contrôleur d'une matrice de disques), l'accès du client suspecté aux disques partagé peut être empêché par un filtrage des requêtes au niveau du SAN (*fabric fencing*).
- Si la grappe dispose d'un commutateur électrique pilotable à distance (*Network Power Switch*), le nœud défaillant est arrêté brutalement par la coupure de son alimentation[52].
- S'il n'est pas possible d'agir sur un nœud à distance, les politiques de contrôle des volumes (stockées sur les disques partagés) sont modifiées pour bloquer les accès effectués par le nœud défaillant. Cette méthode ne garantit pas une protection totales des données car le client fautif a toujours la possibilité d'accéder physiquement aux données s'il outrepasse les permissions du GVLG.

[52] Ce mécanisme est également connu sous l'appellation très imagée de *STO{M/N}ITH* pour « Shoot The Other {Machine / Node} In The Head ».

Par soucis de concision, nous ne détaillerons pas la mise en œuvre des GVLG. Plusieurs implémentations de GVLG sont aujourd'hui disponibles, sous forme de logiciel libre (LVM2/CLVM [183], EVMS [110]) ou de produits commerciaux (CVM [225], SLVM [95]).

2.3.4 Bilan

Cette section a détaillé trois outils logiciels de niveau d'interface « blocs » utilisés pour les E/S dans les grappes de machines : les PGB distants (PGBD), les disques virtuels répartis (DVR) et les gestionnaires de volumes logiques pour grappe (GVLG). Ces mécanismes ne sont pas incompatibles, bien au contraire. Ainsi, un DVR peut être déployé à l'aide de plusieurs instances d'un PGBD. Par ailleurs, les DVR et les GVLG s'appuient souvent sur les mêmes services (gestionnaire de grappe) pour détecter et réagir aux pannes. Toutes les compositions ne sont cependant pas possibles. Par exemple, l'architecture de volumes virtuels définie par Petal s'intègre mal à un GVLG « classique » car elle empiète sur ses fonctions de gestion de volumes.

2.4 Niveau « fichiers »

2.4.1 Introduction

Les systèmes de gestion de fichiers, qu'ils soient centralisés ou répartis, ont toujours occupé une place prépondérante au sein de la thématique des systèmes d'exploitation. L'abondance de l'offre, tant au niveau des solutions industrielles qu'à celui des prototypes de recherche est telle qu'il n'est pas envisageable de fournir une présentation complète des travaux du domaine en l'espace de quelques pages. L'objectif de cette section est beaucoup plus modeste : présenter les principales solutions présentant une interface de niveau « fichiers » déployées actuellement dans le contexte des grappes de machines.

En particulier, nous n'évoquerons pas la « famille AFS » (AFS [100], Coda [122], Intermezzo [32]). Ces travaux, démarrés pour la plupart à Carnegie-Mellon University (CMU), ont apporté des contributions significatives au thème des systèmes de fichiers répartis[53] mais ne sont que peu utilisés dans le contexte des grappes. Historiquement, AFS a été développé au début des années 80 pour fournir une solution répartie d'accès aux données viable à l'échelle d'un campus (et donc d'un MAN — *Metropolitan Area Network*). Coda et Intermezzo sont des prolongements de ce travail optimisés pour des contraintes typiques de ce genre d'environnement (partitions de réseau, déconnexions — volontaires ou non — des clients...). En conséquence, les préoccupations principales de cette famille de systèmes sont axées autour de considérations telles que la duplication optimiste de données, la gestion des déconnexions, la réconciliation de copies divergentes, etc. Ces contraintes (y compris, dans une certaine mesure, les aspects liés à la sécurité) ne s'appliquent que modérément au contexte des grappes. Elles sont en revanche primordiales dans le cadre des grilles de calcul (*Grid Computing*) et des adaptations de Coda ont d'ailleurs été réalisées à cet effet (par exemple [144]). En outre, cette famille de systèmes n'a pas été conçue pour les cas d'utilisation intensive (très grosses charges d'E/S, accès concurrents en écriture vers les mêmes données)[54]. Nous n'aborderons pas davantage ce type

[53] Cette « famille AFS » est d'ailleurs désignée par certains sous l'étiquette (trop floue à nos yeux) des *Distributed File Systems*. Dans ce document, nous employons l'expression « SGF réparti » au sens le plus générique : un système permettant à plusieurs clients s'exécutant sur différents nœuds d'accéder à un ensemble de fichiers

[54] On pourra objecter à ce dernier argument que le système NFS, décrit en 2.4.2.1, présente les mêmes lacunes. Il est cependant beaucoup plus largement utilisé que la famille AFS dans le contexte des grappes de machines, notamment en raison de sa grande simplicité de déploiement. C'est donc aussi pour témoigner de cet « état de fait » que nous avons choisi de faire cette discrimination entre AFS et NFS.

de SGF réparti dans la suite de ce document[55].

Il convient également de préciser que la terminologie et les classifications communément usitées dans le cadre des systèmes de fichiers répartis laissent relativement à désirer[56]. En voici quelques exemples :

- Il n'existe pas d'« esperanto » ni de véritable consensus pour désigner différentes classes de SGF et le sens attribué à une même expression peut varier selon les auteurs.
- Il est souvent question d'une dualité entre topologies SAN (*Storage Area Network*, voir 2.2) et NAS (*Network Attached Storage*, voir 2.4.2) alors qu'il s'agit de deux niveaux d'interface différents (respectivement blocs et fichiers) et que ces deux architectures peuvent être combinées (voir 2.4.4.1).
- De façon assez similaire, il est souvent choisi de classer les systèmes de gestion de fichiers répartis en deux catégories : un modèle où les clients ont un *accès direct* aux supports de stockage et un modèle *à messages* de type client-serveur. Cette présentation a également ses limites : une architecture à accès direct peut s'appuyer sur des organes de stockage partagés au niveau physique mais également sur un disque virtuel réparti. Dans ce second cas, cette sous-couche logicielle peut être vue comme un système à messages qui est, certes, distinct du SGF mais qui peut éventuellement y être fortement couplé (voir par exemple la collaboration entre Petal et Frangipani en 2.4.4.4).

Enfin, les axes de classification sont potentiellement très nombreux, qu'il s'agisse de considérations architecturales (y-a-t-il un point d'accès central au données, le stockage de données et des méta-données sont-ils couplés, etc.) ou fonctionnelles (sémantique de cohérence des caches, mise en œuvre du verrouillage...). La classification choisie dans cette section est par conséquent discutable. Nous nous efforcerons cependant d'aborder tous ces critères principaux.

Le survol que nous proposons est divisé en trois catégories.

- Les systèmes « clients-serveur » permettent à plusieurs clients d'accéder aux fichiers stockés sur un volume local d'un nœud serveur.
- Les SGF parallèles gèrent explicitement, au niveau de chaque client, la distribution des données sur les disques locaux de plusieurs nœuds afin de maximiser le débit d'E/S.
- Les SGF partagés, conçus avant tout pour la haute disponibilité, reposent sur le fait que tous les clients peuvent accéder directement à un espace de stockage, physique (SAN) ou logique (DVR), de bas niveau (bloc ou objet).

Enfin, nous présentons en 2.4.5 l'état courant des travaux sur les SGF répartis basés sur l'interface de stockage à objets.

2.4.2 Systèmes clients-serveur

Le principe de cette famille de SGF répartis peut être résumé de la sorte : un serveur permet à des clients distants d'accéder à une arborescence locale de manière transparente. On peut envisager ces systèmes de fichiers basés sur l'export de volumes comme des « méta-SGF », c'est à dire la mise en œuvre d'un ensemble de protocoles permettant d'exploiter des ressources centralisées (couche VFS du client, SGF local du serveur) dans un contexte réparti.

[55] Voir par exemple [164], pour un survol moins restrictif de la thématique du stockage réparti.

[56] Des tentatives de solutions sont cependant en cours d'ébauche, tel le *Shared Storage Model* [204] défini par le consortium SNIA (*Storage Networking Industry Association*), qui se veut être le pendant du modèle architectural OSI défini pour les réseaux.

2.4.2.1 Network File System (NFS)

NFS est un protocole développé initialement par Sun Microsystems puis normalisé par l'*Internet Engineering Task Force* (IETF) ; ses objectifs principaux sont la simplicité et l'interopérabilité. Il en existe aujourd'hui trois versions publiques, numérotées de NFSv2 [214]à NFSv4[57]. Les versions 2 et 3 sont assez similaires ; la dernière est radicalement différente. Un dénominateur commun pour toutes ces versions est l'emploi d'appels de procédure à distance (ou *RPC* pour *Remote Procedure Call* [213]) et de l'encodage XDR (*External Data Representation* [212]).

NFSv2 Il s'agit d'un protocole sans état : chacune des requêtes émises par un client contient toutes les informations nécessaires pour que le serveur puisse y répondre. Ainsi, il n'est pas nécessaire que le serveur conserve des informations sur l'état des clients (par exemple les fichiers en cours d'utilisation). Cet aspect « sans état » simplifie l'implémentation des différents modules NFS ainsi que la gestion des pannes (du serveur, du lien de communication)[58]. En contrepartie, il a un impact négatif sur les performances et la gestion de la concurrence.

Pour limiter le coût des accès distants, chaque client NFSv2 utilise le cache local pour stocker les données lues. La cohérence des caches, assez relâchée, est basée sur un modèle temporel. Un client peut garder des informations (données et méta-données) lues pendant quelques secondes avant de devoir les recharger auprès du serveur (en général 3 secondes pour les données et 1 minute pour les méta-données). Les données écrites peuvent être conservées jusqu'à 30 secondes par le client avant d'être envoyées au serveur (et sont également synchronisées lors de la fermeture d'un fichier). NFSv2 ne fournit donc pas de fortes garanties de cohérence en cas d'accès concurrent aux données. Le serveur n'est pas autorisé à retarder l'écriture des données (NFSv2 impose une politique *write-through*).

NFSv2 fournit un service de verrouillage optionnel. Celui-ci n'est pas intégré au protocole NFS proprement dit car ce serait contradictoire avec sa nature « sans état ». Le verrouillage est donc assuré par le protocole auxiliaire Network Lock Manager (NLM) [162], qui permet une granularité variable (fichier complet ou plage d'octets), et des interactions synchrones ou asynchrones entre le serveur et les clients.

NLM fournit deux types de verrous : les *verrous supervisés* et les *verrous normaux*. Contrairement aux verrous normaux, les verrous supervisés fournissent plusieurs garanties en cas de panne : lors du redémarrage d'un serveur après une panne, ces derniers sont automatiquement restaurés, sans nécessiter d'interactions avec les applications clientes. En outre, en cas de panne d'un client, les verrous supervisés qu'il détient sont automatiquement libérés lors du redémarrage du client. Pour fournir ces garanties, NLM s'appuie sur un autre service, *Network Status Monitor* (NSM) qui permet à un nœud d'informer d'autres nœuds (via la diffusion ponctuelle d'un simple numéro de séquence) que son état vient de changer (*état incohérent* lors du redémarrage après une panne, *état cohérent* lorsque les mesures nécessaires ont été prises). Comme on peut le voir, les garanties du système de verrouillage sont assez faibles et ne prennent pas en compte tous les cas de figures (client ne redémarrant pas alors qu'il détient des verrous, partition de réseau, etc.). En outre, le protocole de verrouillage/reprise est basé sur l'hypothèse implicite que les canaux de communications sous-jacents sont fiables et respectent l'ordre, ce qui n'est pas forcément le cas (il est possible d'utiliser UDP).

Par ailleurs, il est important d'insister sur le caractère *optionnel* des verrous : puisque le serveur NFS n'est pas conçu pour gérer l'attribution et la vérification des verrous, celui-ci présume que tous

[57] La version 1 n'était qu'un prototype interne de Sun Microsystems.

[58] Un strict modèle sans état peut cependant poser quelques problèmes et la plupart des implémentations le transgressent quelque peu, en particulier pour une meilleure gestion des opérations non idempotentes. Par exemple, si le protocole de transport ne garantit pas une délivrance fiable des messages (cas d'UDP), un acquittement de suppression de fichier peut se perdre. Ceci conduit le client à réitérer sa demande et le serveur lui retournera une erreur car le fichier n'existe déjà plus. Pour éviter ce problème, de nombreux serveurs gardent un cache des dernières opérations effectuées.

les clients « jouent le jeu » en collaborant avec le serveur NLM avant d'accéder à des données, ce qui n'est pas forcément le cas en pratique.

NFSv2 peut être déployé au dessus des protocoles UDP/IP ou TCP/IP. La nature « sans connexion » d'UDP est bien adaptée au fait que NFSv2 ne maintient pas d'état sur le serveur. En conséquence, de nombreuses implémentations de NFSv2 sont basées par défaut sur UDP. Cependant, dans le cas d'utilisations assez intensives, le protocole de transport UDP, qui n'effectue aucun contrôle de flux, pose des problèmes de performance significatifs. Dans ces conditions, de nombreux utilisateurs optent pour la version TCP, qui offre de moins bonnes performances « à vide » mais permet de tolérer plus harmonieusement les pics de requêtes.

NFSv3 Cette version [171, 38] ne présente pas de grandes différences architecturales avec la précédente ; elle introduit un ensemble d'optimisations, dont notamment :

- la gestion d'opérations composées sur les méta-données (pour limiter les transferts sur le réseau et améliorer la latence des opérations),
- le support des tailles de fichiers sur 64 bits,
- la négociation de la taille des segments de données échangés entre le client et le serveur pour optimiser les transferts (NFSv2 la limite à 8 ko, en version 3, elle est généralement de 32 ko).
- la gestion des écritures asynchrones (un serveur peut acquitter des requêtes d'écritures sans les avoir répercutées sur disque ; le client peut forcer le vidage du cache serveur sur disque par une requête spéciale).

NFSv4 Cette version [170, 39] est une refonte profonde du protocole, qui tente de remédier à ses lacunes les plus notoires. Les changement majeurs sont l'ajout de la notion d'« état » au protocole et l'imposition de TCP/IP comme mode de transport.

Le modèle de cohérence a également été amélioré par le recours à des *baux* (*leases*) et à des *délégations* (*share reservations*). Un bail fournit à un client une garantie temporaire sur la cohérence d'une donnée en lui conférant un verrou périssable. Un bail en lecture permet à un client de garder une copie en lecture sans crainte de modification concurrente. Un bail en écriture permet à un client de retarder l'écriture de données sur le serveur sans compromettre la vision que d'autres clients ont de ces données. Un client doit périodiquement renouveler ses baux (un seul message pour tous les verrous d'un client). Ce protocole permet également de détecter la panne d'un client, et le cas échéant de libérer les verrous qu'il détenait ; il simplifie également la reprise après la panne d'un client ou du serveur. Une délégation permet à un client de gérer lui même la concurrence des accès à un fichier (entre plusieurs entités locales) s'il en est le seul utilisateur. Le verrouillage, intégré à NFSv4, est de surcroît *obligatoire*, dans le sens où un verrou ne peut être contourné comme c'était le cas précédemment.

NFSv4 est également capable de gérer la migration et la duplication de données. Si les données exportées sont déplacées d'un serveur à un autre, un client peut récupérer le nouvel emplacement des données auprès de l'ancien serveur et continuer à fonctionner sans perturber les applications concernées. Pour les données en lecture seule, des serveurs subsidiaires peuvent être spécifiés et utilisés par le client en cas de panne ou de mauvaises performance du serveur principal.

D'autres améliorations concernent la gestion des opérations composées (plusieurs requêtes agrégées dans un seul appel RPC), ainsi qu'une interopérabilité renforcée (les précédentes versions étant implicitement très dépendantes du modèle Unix de protection et d'accès aux données). Pour finir, le protocole a été très nettement sécurisé (les versions précédentes laissant beaucoup à désirer en la matière), au niveau de l'authentification des utilisateurs comme à celui de la confidentialité des données. La politique de sécurité est en outre négociable dynamiquement.

A l'heure actuelle, il n'existe que peu d'implémentations (peu matures voire incomplètes) de cette dernière version du protocole NFS. Par conséquent, elle ne dispose pas d'une grande base d'utilisateurs et une étude réaliste et équitable de ses performances n'est pas encore envisageable. Quoi qu'il en soit, ces modifications font de NFS une solution beaucoup plus adaptée aux contraintes de l'Internet que par le passé, et le prédestinent à de nouveaux rôles (sauvegarde distante de données personnelles, protocole de téléchargement optimisé, etc.). Dans le contexte des grappes, l'amélioration du verrouillage, de la gestion de la cohérence et la prise en compte des données déplacées ou dupliquées (en lecture seule) sont intéressantes. Il n'en demeure pas moins quelques problèmes importants pour certaines applications ayant des contraintes particulières (lourdes charges d'E/S, accès concurrents en écriture ou encore besoins en terme de haute disponibilité), le principal étant le serveur, qui constitue un goulot d'étranglement et un point central de défaillance.

2.4.2.2 Common Internet File System (CIFS)

Ce protocole est développé par Microsoft depuis une vingtaine d'années [192]. En termes de popularité, CIFS (anciennement SMB pour *Server Message Block*) est le pendant de NFS pour les systèmes d'exploitation Windows. Il en diffère toutefois significativement, d'une part car sa portée dépasse le strict cadre des systèmes de fichiers répartis (il permet d'exporter et de partager d'autres types de ressources et de services tels que l'impression), d'autre part car son fonctionnement est assez différent de NFS (versions 2 et 3). L'analogie avec NFSv4 est plus pertinente même si des différences subsistent, par exemple la gestion des caches clients qui peut éventuellement, à l'instar de Coda, supporter les déconnexions.

CIFS, développé avant tout pour les systèmes Microsoft, n'est que marginalement utilisé dans le cadres des grappes qui sont pour la plupart exploitées sous Unix/Linux.

2.4.2.3 Serveurs multiprotocoles spécialisés (NAS)

La notion de « Network-Attached Storage » (NAS), bien que vidée de son sens par une utilisation outrancière et souvent hors de propos, désigne généralement les serveurs exportant leurs données via des protocoles tels que NFS et CIFS. Il existe une importante gamme de solutions matérielles et logicielles spécialisées pour ces usages : on parle de *NAS heads*, *NAS appliances* ou encore de *NAS filers*.

Ces solutions se distinguent par les caractéristiques suivantes :

- La machine est uniquement consacrée au rôle de serveur de fichiers et ses ressources matérielles sont précisément dimensionnées en conséquence. Le stockage est généralement basé sur une matrice RAID pour tolérer les pannes et accroître les performances.
- Des optimisations matérielles sont fréquentes, par exemple le recours à une mémoire non volatile (NVRAM) pour accélérer les acquittements des écritures par le serveur.
- Le système de fichiers peut être optimisé pour les caractéristiques typiques des charges d'E/S auxquelles il est destiné. En outre, le système d'exploitation peut également être optimisé pour son rôle unique de serveur de fichiers (par exemple, absence de mode utilisateur pour éviter le coût des changements de contexte).
- Enfin, une couche logicielle permet de rendre la sémantique de cohérence et de verrouillage uniforme pour le serveur malgré les différences significatives entre les protocoles utilisés par les clients (par exemple NFSv3 et CIFS) [30].

En quelques mots, il s'agit donc d'une implémentation optimisée et flexible (plusieurs protocoles sont supportés) d'un serveur de fichiers centralisé fonctionnellement semblable aux systèmes décrits ci-avant.

2.4.2.4 Systèmes clients-serveur optimisés

Un certain nombre de projets ont cherché à améliorer les performances des SGF clients-serveur en jouant principalement sur deux axes : l'emploi de réseaux optimisés et le recours à plusieurs serveurs d'E/S.

Optimisations pour réseaux spécialisés La première « famille » de travaux étudie l'utilisation de réseaux optimisés (fournissant par exemple une interface VIA) pour améliorer les performances des systèmes clients-serveur. L'exemple le plus simple consiste à adapter la couche de transport d'une implémentation NFS à une interface basée sur des transferts par RDMA [37]. Ceci permet de bénéficier des caractéristiques avantageuses offertes par le matériel employé (faible latence, haut débit, gestion matérielle des erreurs de communication) et de décharger les clients et surtout le serveur de la consommation CPU inhérente à la pile TCP/IP.

D'autres approches, telles que DAFS [59, 141] et ORFA [85], vont plus loin en cherchant à éliminer toutes les copies de données habituellement nécessaires lors d'un transfert entre client et serveur. Le prototype ORFA (*Optimized Remote File System Access*) cherche à exploiter au mieux les réseaux Myrinet sous Linux, en masquant le stockage distant des données aux applications par le biais d'interceptions des appels système.

DAFS (*Direct Access File System*) est un standard industriel émergeant basé sur NFSv4. Il en diffère cependant assez largement, notamment par les points ci-dessous.

- Il est intrinsèquement conçu pour des réseaux capables de RDMA.
- Le contrôle de flux des transferts de données est géré uniquement au niveau du serveur. Par exemple, une requête d'écriture envoyée par un client ne contient pas de données ; le serveur démarre en conséquence un ou plusieurs transferts en lecture vers le client. Le serveur est ainsi l'entité centrale qui dispose de tous les leviers pour ordonnancer et cadencer au mieux les différents flux de données [158].
- Le client DAFS est implémenté au niveau utilisateur (comme bibliothèque liée aux applications) pour fournir une portabilité optimale et bénéficier des fonctionnalités d'*OS bypass* des cartes réseau optimisées.
- Le point précédent permet en outre une optimisation supplémentaire : les applications clientes peuvent bénéficier d'E/S asynchrones même si le système d'exploitation de l'hôte ne fournit pas une telle fonctionnalité. DAFS est optimisé pour les applications basées sur des requêtes d'E/S non bloquantes, qui permettent de gérer les opérations de manière « pipelinée ».
- DAFS n'utilise pas de cache au niveau des clients car il est principalement destiné à des applications gérant elle même leur politique de cache/préchargement de données et ne nécessitant pas de partage inter-processus. Même s'il est compatible avec un usage généraliste, DAFS est principalement développé comme un substrat élémentaire pour des bibliothèques d'E/S (scientifiques, mini bases de données, ...) simplifiant le développement d'applications.

Répartition La deuxième famille d'améliorations est fondée sur la répartition transparente des données sur plusieurs serveurs. Il s'agit ainsi d'une approche hybride entre SGF clients-serveur et SGF parallèle : à la différence du second cas, les clients ne dialoguent qu'avec une seule entité, le serveur principal. La quasi-intégralité des travaux dans ce domaine ont produit des prototypes basés sur NFS, en raison de sa popularité et de l'abondance de ses implémentations.

Le système Mirage [21] agrège plusieurs serveurs NFS(v2) usuels et présente aux clients un serveur « virtuel », qui a pour rôle de router les requêtes entre le client et le serveur NFS concernés. Le routeur réexporte ainsi les différents volumes mis à disposition par chacun des serveurs et effectue

également un arbitrage de qualité de service, afin de partager la bande passante équitablement entre les différents clients (et de lutter éventuellement contre les attaques de type « déni de service »). La répartition de charge est implicite et présuppose que le placement et l'accès aux données par les clients sont effectués de manière relativement uniforme sur les différents volumes.

A l'instar de Mirage, le projet Cuckoo [123] utilise des clients et serveurs NFS(v2) usuels et agrège plusieurs volumes grâce à un mandataire situé entre les clients et le serveur[59]. Sa particularité est de répondre aux problèmes de montée en charge, en dupliquant les fichiers fréquemment lus et pas (ou peu) modifiés. Ce clonage des données est en outre effectué lors des périodes d'oisiveté des serveurs, afin de ne pas contribuer à l'écroulement du système.

Le prototype NFSp [135, 164], développé dans le cadre des grappes et des grilles de calcul bon marché, vise quant à lui à paralléliser l'intégralité des E/S consécutives à une requête NFS(v2). Le serveur central ne stocke que les métadonnées associées à un fichier et convertit les requêtes NFS liées à son contenu en une série de requêtes d'E/S vers plusieurs nœuds, nommés *iods* (pour « I/O daemons »), sur lesquels les données sont réparties (par *striping*). Dans le cas d'une lecture de fichier, un iod répond directement à un client en modifiant les paquets UDP émis pour « imiter » le serveur central[60]. Des travaux ultérieurs ont étudié la duplication du serveur central ainsi que des données pour tolérer certains cas de pannes sans perturber les clients. La haute disponibilité n'est cependant pas un objectif majeur du système et sa résilience est limitée. NFSp a été validé dans de nombreux contextes liés au calcul scientifique, sur grappe et sur grille de grappes. Outre les les limitations inhérentes à NFS(v2), cette approche souffre cependant de sa volonté de transparence auprès des clients pour les performances en écriture, car toutes les données transitent obligatoirement par le serveur central, qui reste ainsi un goulot d'étranglement.

L'architecture Slice [10] se présente comme une solution de migration progressive d'un modèle clients-serveur classique (utilisant du matériel banalisé pour LAN) vers une infrastructure à hautes performances, basée sur un réseau à haut débit et des baies de stockage partagées par plusieurs serveurs. Comme dans les exemples précédents, l'interface présentée aux clients (NFSv3) n'est pas modifiée, grâce à l'emploi d'un module d'interception. Ce dernier (nommé μproxy) peut être placé à différents endroits du réseau, par exemple sur un commutateur. Dans le cas du prototype développé à Duke University, le μproxy est inséré dans la pile IP de chaque client et intercepte les paquets échangés avec un serveur virtuel (reconnu par son adresse IP). Contrairement aux exemples précédents, le code des serveurs est spécifique à l'architecture et ne permet pas la réutilisation d'un serveur NFS « classique ». En fonction du type de requête NFS, le μproxy détermine vers quel ensemble de serveurs la requête doit être redirigée, puis, en fonction des arguments, quel serveur précis doit être contacté. Le serveur concerné est alors chargé de la manipulation des informations (données et/ou métadonnées) sur les baies de disques, celles-ci étant stockées selon un modèle à objets (OSD). Certaines requêtes NFS pouvant impliquer plusieurs serveurs, un service distribué de coordination permet d'assurer l'intégrité des informations, la reprise sur pannes ainsi que l'ajout ou le retrait dynamique de serveurs. Ces aspects sont simplifiés par le fait que les μproxys et (dans une certaine mesure) les serveurs ne maintiennent pas d'état fortement synchronisé sur l'architecture du système et le placement des données. Dans le contexte du prototype implémenté, les serveurs sont divisés en trois ensembles : noms, petits fichiers et E/S lourdes (supérieures à 64 ko). Pour chaque catégories, la politique de choix du serveur peut être différente, tout comme celle de répartition et de duplication

[59] Les mandataires utilisés par Mirage et Cuckoo ont pour principale vocation d'être embarqués dans un commutateur réseau (switch) intelligent. Dans le cas de Cuckoo, la logique de (re)routage et de duplication peut également être déployée sur chacun des nœuds serveurs, en amont du serveur NFS, même si cela présente davantage de difficultés techniques (synchronisation des copies, redirection transparente d'une connexion TCP).

[60] Ceci est beaucoup plus délicat à mettre en œuvre avec un protocole en mode connecté tel que TCP. Une telle approche est donc limitée à un usage en mode datagrammes et peut souffrir d'un mauvais contrôle de flux en cas de congestion du réseau.

des données. De plus, différentes politiques peuvent être appliquées à différents fichiers.

2.4.2.5 Bilan sur les systèmes clients-serveur

De tels systèmes de fichiers distribués sont généralement faciles à mettre en œuvre et à déployer. Ceci s'explique principalement par le fait qu'il s'agit plutôt de « protocoles de glue » que de véritables systèmes de fichiers développés de « fond en comble » (en particulier, le stockage sur disque des informations repose sur les services fournis par le système de fichiers local sur le(s) nœud(s) impliqué(s)). En outre, cette famille de SGF répartis n'a (dans la plupart des cas) pas de pré-requis matériel et des implémentations sont largement disponibles depuis une vingtaine d'années. Elle bénéficie ainsi d'une grande base d'utilisateurs et d'une grande expertise industrielle et académique. Ces SGF ne sont toutefois pas généralistes : ils sont bien adaptés à des charges d'E/S modérées et majoritairement constituées de lectures mais ne fournissent pas les garanties nécessaires (cohérence, performance, disponibilité) à des applications intensives en E/S.

Des travaux ont visé à remédier à une ou plusieurs de ces lacunes (mais jamais l'ensemble) en jouant sur certains des leviers suivants :

- refonte des protocoles pour permettre une cohérence plus forte ainsi que la gestion de données dupliquées (en lecture seule) ou déplaçables ;
- optimisations des transferts de données pour les interfaces de communication à haut débit ;
- emploi d'un module « mandataire » (situé soit au niveau de chaque client, soit à celui d'un serveur central) opérant des fonctions de duplication de données et de répartition de charge, qui fait ainsi tendre un SGF clients-serveur vers un SGF parallèle.

2.4.3 Systèmes de fichiers parallèles

L'épithète « parallèle » a ici un sens bien précis : la portion cliente du SGF gère explicitement la distribution des données sur plusieurs disques[61] afin de maximiser le débit d'E/S. Plus précisément, les données sont stockées sur les disques d'un ensemble de machines participant au service des fichiers.

Nous présentons ici trois exemples de SGF parallèles. Les deux premiers sont optimisés pour les besoins d'un contexte applicatif précis alors que le troisième est généraliste.

2.4.3.1 Parallel Virtual File System (PVFS)

Le projet PVFS, développé à Clemson University et au laboratoire Argonne, a pour objectif de fournir un système de fichiers parallèle pour les grappes sous Linux. Il est conçu en priorité pour les besoins des applications de calcul scientifique manipulant de gros volumes de données. Ces applications ont généralement pour contrainte majeure leur vitesse d'exécution. La haute disponibilité et l'intégrité des données ne sont pas primordiales : des points de reprise sont générés au niveau applicatif, les données manipulées sont généralement facilement reproductibles et/ou archivées sur un support de stockage tertiaire.

Il existe actuellement deux versions de PVFS : la première [40] a été largement diffusée et déployée depuis plus de cinq ans, la seconde [131] est à l'heure actuelle encore au stade de prototype.

[61] Contrairement à certains systèmes décrits en 2.4.2.4, qui répartissent également les données sur plusieurs nœuds.

PVFSv1 L'architecture de PVFS distingue deux types de serveurs : un *manager* et plusieurs *iods*. Le manager est l'interlocuteur principal des clients et il n'en existe qu'une seule instance. Il stocke les méta-données des fichiers (dans des fichiers locaux), décide du placement des données sur les iods et met en relation clients et iods[62]. Un iod a pour rôle de recevoir des requêtes d'E/S de la part des clients et de les satisfaire. Les blocs de données sont stockés sous forme de fichiers dans le SGF local de chaque iod (le nom de chaque fichier est déterminé par une méthode de hachage). Lorsqu'un client manipule un fichier, il envoie sa requête au manager, qui lui retourne les informations nécessaires (notamment la liste des iods à contacter) pour lire/écrire les données. Les communications entre les différentes entités sont basées sur des canaux TCP.

PVFSv1 ne gère pas de cache au niveau des clients, ce qui constitue un frein aux performances. En contrepartie, puisque chaque accès aux données va jusqu'aux iods et que les écritures sont synchrones, chaque client a toujours une vue cohérente des données. PVFSv1 ne fournit pas de système de verrouillage car il suppose que la synchronisation des accès aux fichiers est effectuée au niveau applicatif.

PVFS ne gère aucune redondance des informations, qu'il s'agisse des méta-données ou des données. Les données sont réparties sur les iods selon un schéma de *striping* (principe du RAID-0), l'iod de départ pouvant varier selon les fichiers pour éviter la surcharge d'un iod particulier. Des projets de recherche ont cherché à améliorer la tolérance aux fautes et la flexibilité du système en dupliquant toute l'architecture selon un schéma RAID-10 ainsi qu'en faisant varier le schéma de duplication des données (RAID-1 ou RAID-5 en fonction des caractéristiques de chaque fichier) [239, 174].

Les premières implémentations reposaient sur des démons en mode utilisateur (pour les trois types d'entités), ce qui induisait de nombreuses recopies de données superflues. Les dernières versions ont répondu à ce problème en fournissant une implémentation en mode noyau.

PVFSv2 Cette seconde version met l'accent sur la configurabilité du système. Les principales évolutions sont les suivantes :

- Le support de plusieurs interfaces de communication : il n'est plus obligatoire d'utiliser des sockets TCP pour l'interaction entre clients, managers et iods. Des modules optimisés pour certaines interfaces (génériques ou spécifiques à une famille de matériel telle qu'Infiniband) peuvent être mis en œuvre.
- Le schéma de répartition des donnés sur les iods est configurable, il est possible de développer des stratégies optimales pour les comportements particuliers de certaines applications scientifiques.
- La gestion des méta-données peut être répartie sur plusieurs managers. Cette possibilité n'a cependant qu'un intérêt limité car en pratique, un manager n'est pas un point de contention.
- La sémantique de cohérence est paramétrable (sans toutefois pouvoir respecter la norme POSIX, jugée trop contraignante et peu utile dans le cadre des applications scientifiques à supporter). La documentation actuelle du SGF manque néanmoins de précisions à ce sujet.
- Enfin, il est également prévu de permettre la duplication paresseuse des données et méta-données mais ce axe de développement n'est qu'embryonnaire.

2.4.3.2 Google File System

Ce SGF réparti récent (présenté en 2003 [81]) a quelques similitudes avec PVFS mais présente également plusieurs caractéristiques atypiques découlant de son contexte d'utilisation. Il a été conçu

[62] Il est à noter que ces différentes entités (clients, manager et iods) sont logicielles et qu'un même nœud physique peut en héberger plusieurs types ; ce cas est cependant rare en pratique.

pour les besoins de la société Google sur la base des hypothèses suivantes :

- Les grappes visées sont constituées de 100 à 1000 PC bon marché sous Linux, reliés par un réseau Fast Ethernet. De ce fait, des pannes surviennent quotidiennement et doivent être prises en compte.
- Les applications clientes effectuent principalement des tâches d'indexation de documents. Celles-ci sont toutes développées au sein de la compagnie et sont exclusivement à usage interne. Il n'est donc pas impératif de fournir une interface parfaitement compatible avec les standards (par exemple l'API POSIX) et d'intégrer le système de fichiers à la couche VFS.
- Les applications manipulent quelques millions de gros fichiers (de 100 Mo à plusieurs Go) et nécessitent un débit soutenu d'accès au données. Elles n'ont en revanche pas de grosse contrainte au niveau de la latence des opérations. Il n'y a pas de réel besoin de cache de données au niveau des clients, qui effectuent généralement un unique accès séquentiel à un fichier ou manipulent des quantités de données très supérieures à la taille de la mémoire centrale.
- Les principaux types d'accès aux données sont assez particuliers : grands transferts en lecture ou écriture séquentielle (1 Mo ou plus) et lectures aléatoires de petites quantité de données. De plus, les écritures sont presque toujours effectuées en fin de fichier.
- Enfin, les besoins en matière de synchronisation sont limités. En particulier, une stricte sémantique POSIX s'avère superflue. Un besoin important concerne toutefois les situations où plusieurs clients écrivent de façon concurrente à la fin d'un fichier lu simultanément par un ou plusieurs clients.

L'architecture de GoogleFS est basée sur trois types d'entités logicielles schématiquement assez proches de celles de PVFS : *clients*, *master* et *chunkservers*. Chaque entité est implémentée sous forme d'un processus utilisateur communiquant avec les autres via TCP/IP. Les fichiers sont divisés en unités de taille fixe (64 Mo) nommées *chunks*, allouées par le master et stockées par les chunkservers ; la granularité de manipulation de données par les clients reste bien sûr l'octet. Chaque chunk, désigné par un identificateur global (unique et immuable), est dupliqué sur plusieurs chunkservers (3 par défaut). Le master est en charge de la gestion des méta-données, de la cohérence des accès et de la surveillance des chunkservers.

Pour accéder au contenu d'un fichier, un client commence par contacter le master pour récupérer les méta-données nécessaires (qu'il peut garder temporairement en cache), puis peut communiquer directement avec un chunkserver pour les transferts de données. Le modèle de cohérence est relâché mais permet de satisfaire les besoins des applications. GoogleFS est en fait implementé sous forme d'une bibliothèque applicative liée aux applications. Ceci permet l'introduction de quelques opérations supplémentaires dont une fonctionnalité d'*append* atomique et une autre de *snapshot*. L'opération append garantit qu'un enregistrement est ajouté (sans entrelacement avec d'autres données) au moins une fois à la fin d'un fichier. Pour décharger le master, la responsabilité de la synchronisation des exemplaires d'un même chunk est confiée à un chunkserver. Différents exemplaires d'un même chunk peuvent avoir un contenu différent (par exemple certains exemplaires peuvent contenir des doublons). Les programmeurs d'application doivent par conséquent prendre certaines précautions dans leur code mais leur impact est modéré. Le mécanisme de snapshot est géré efficacement et assez facilement grâce à la nature centralisée du service de méta-données et aux techniques habituelles de *copy-on-write*.

L'implémentation du master présente en outre certaines fonctionnalités et optimisations supplémentaires. Les méta-données décrivant le contenu d'un fichier sont stockées sur disque de manière journalisée mais également intégralement conservées en mémoire pour accélérer le traitement des requêtes. La correspondance entre un identifiant de chunk et les serveurs qui en stockent une copie

n'est pas gardée de manière permanente par le master. Cette liste est reconstituée à chaque redémarrage du système par un dialogue très rapide entre le maître et les serveurs. Le master peut également décider d'ajouter de nouveaux exemplaires d'un chunk (pour maintenir ou augmenter le taux de redondance) ou d'en déplacer certains pour améliorer les performances du système. Même si le système de nommage exposé aux applications est indépendant de la localisation des données, GoogleFS prend en compte ces informations. Ainsi, un client choisit généralement le chunkserver le plus proche de lui[63]. De plus, le master tend a choisir des chunkservers éloignés les uns des autres pour le stockage d'un chunk donné, afin de fournir des performances d'accès équitables à l'ensemble des clients, mais aussi pour éviter les scénarios catastrophe (tels qu'une anomalie électrique paralysant toutes les machines d'un même rack). Enfin, le master maintient à jour des copies de ses méta-données sur d'autres nœuds. Si une panne du master est détectée (par un administrateur humain ou un service de supervision de la grappe), celui-ci peut être rapidement redémarré ou remplacé, de manière transparente pour les clients et les chunkservers (modification d'alias DNS).

En pratique, les performances sont satisfaisantes (le débit des écritures reste toutefois perfectible) et le système passe à l'échelle. En outre, le serveur central de méta-données n'est pas un goulot d'étranglement pour les performances.

GoogleFS est donc un exemple réussi de SGF réparti basé sur des ressources matérielles banalisées et des optimisations poussées pour un contexte d'utilisation particulier.

2.4.3.3 xFS

Le système xFS[64] a été développé à l'université de Berkeley au milieu des années 1990 [11]. L'idée motrice du projet est d'éviter d'architecturer un SGF réparti autour d'entités centralisées pour échapper aux contraintes classiques (goulot d'étranglement et sensibilité aux pannes)[65], tout en fournissant aux clients une cohérence forte pour l'accès aux données. Cette motivation était de surcroît renforcée par l'émergence de nouveaux réseaux très performants tels qu'ATM et Myrinet.

L'architecture de xFS définit quatre types d'entités : clients, gestionnaires de méta-données (*managers*), serveurs de données et nettoyeurs (*cleaners*). Chaque nœud de la grappe endosse généralement les quatre rôles.

Chaque manager est responsable de la gestion d'un ensemble de fichiers, la liste des fichiers affectés à un manager donné peut évoluer au cours du temps, pour réagir à l'ajout ou au retrait de nœuds ainsi qu'à d'éventuels problèmes de performances. Un manager est notamment chargé d'assurer la mise à jour des méta-données et la cohérence de l'accès aux données. Des heuristiques tentent d'optimiser l'association entre fichiers et managers, par exemple en co-localisant sur le même nœud le manager d'un fichier et le client qui l'a crée.

Chaque fichier est lui-même associé à un groupe de serveurs de stockage (SG pour *Stripe Group*) sur lequel les données sont étalées selon un schéma de distribution particulier (en général à la RAID-5, avec un ou plusieurs segment de parité). Utiliser plusieurs SG permet notamment d'optimiser les E/S concurrentes sur différents fichiers et d'harmoniser la charge. De plus, si la grappe est d'assez grande taille, répartir les données de chaque fichier sur l'ensemble de nœuds n'est généralement pas

[63] Le réseau étant organisé selon une cascade de commutateurs, le temps de communication entre deux nœuds est variable selon leur proximité.

[64] Attention, xFS n'a rien à voir avec XFS, un système de fichier centralisé, ni avec la version partagée de ce dernier (cf 2.4.4.3), tous deux développés par la compagnie SGI.

[65] Les premiers travaux autour de xFS concernaient plutôt les environnements à moyenne et large échelle, et utilisaient en conséquence des stratégies agressives de cache au niveau de clients ainsi qu'une hiérarchie de serveurs de méta-données. Cette optique a été assez rapidement abandonnée et l'architecture du SGF a été largement revue pour le contexte des réseaux locaux à hautes performances.

une approche efficace. De nouveaux SG peuvent être créés au fil du temps et d'autres considérés « obsolètes », ce qui permet de réagir à l'ajout ou au retrait de nœuds.

Chaque manager maintient les informations nécessaires pour assurer la cohérence d'un fichier. En particulier, il garde une liste de tous les clients possédant une copie d'un bloc donné dans leur cache. Une cohérence forte des données est assurée à l'aide d'un jeton distribué pour chaque bloc de données. Avant d'autoriser un client à modifier le contenu d'un bloc, le manager en invalide toutes les copies présente dans les caches clients. Un client conserve le droit de modifier un bloc dans son cache jusqu'à ce qu'un autre client demande à y accéder. Les informations d'état d'un manager permettent ainsi de mettre en œuvre un mécanisme de *cache coopératif* : lorsqu'un client (A) cherche à lire un bloc de données, si le manager correspondant détecte qu'un autre client (B) possède une copie du bloc, alors la requête est transmise à B qui répondra directement à A.

Afin d'optimiser les accès aux serveurs de stockage, xFS repose sur une stratégie très largement inspirée de l'un de ses prédécesseurs, Zebra [92]. Cette méthode consiste de façon schématique à combiner le principe du RAID logiciel à celui des LFS (cf 2.1.2.3) ; elle permet notamment de remédier aux mauvaises performances d'un schéma de redondance par parité (tel que RAID-5) pour les petites écritures et la reconstruction d'un disque. Chaque client écrit ses données dans un tampon local suivant le modèle LFS, ce qui permet d'agréger les écritures. Les informations écrites sont ensuite fragmentées (avec calcul d'un fragment de parité) et envoyées aux différents serveurs de stockage du SG concerné. Cette méthode fournit de bonnes performances d'accès aux données. Elle nécessite cependant une détection et une récupération perpétuelle de l'espace libéré sur disque au fil des modifications de leur emplacement. Contrairement à Zebra, xFS gère cette tâche de manière répartie avec un ensemble de « nettoyeurs », ce qui s'est avéré difficile à implémenter efficacement.

Le prototype xFS a été implémenté en mode noyau, basant ses communications sur des canaux TCP/IP. Il a été déployé sur une grappe d'une trentaine de nœuds interconnectés par un réseau Myrinet.

Le bilan de xFS est mitigé. Il s'agit d'un projet pionnier à de nombreux égards : suppression de toutes les entités centralisées sans recours à du matériel spécialisé (autre qu'un réseau rapide), reconfiguration dynamique, validation des techniques de cache coopératif et meilleur passage à l'échelle qu'AFS ou NFS tout en fournissant une cohérence forte. Plusieurs aspects ne sont cependant pas entièrement satisfaisants. Tout d'abord le fonctionnement global du système est très complexe et ses concepteurs ont d'ailleurs peiné à mettre en œuvre toutes les fonctionnalités. L'implémentation d'une telle architecture est dure à maintenir et a fortiori à étendre. De plus, les performances absolues du système sont relativement limitées. Enfin, les mécanismes de reprise sur panne et de reconfiguration dynamique sont complexes et coûteux, ce qui constitue un frein aux objectifs de passage à l'échelle et de haute disponibilité.

2.4.3.4 Bilan sur les systèmes de fichiers parallèles

Les systèmes de fichiers parallèles sont basés sur l'emploi de plusieurs nœuds serveurs explicitement exposés à la vue des clients. Une telle architecture peut offrir des bénéfices substantiels en termes de passage à l'échelle (en nombre de clients, taille des requêtes et charge globale du système) en raison du parallélisme d'E/S ainsi qu'une meilleure robustesse (si les données sont dupliquées sur plusieurs nœuds de stockage). Ce type de SGF réparti s'avère bien adapté pour des applications aux besoins modestes en termes de fonctionnalités de verrouillage et de sémantique de cohérence ; c'est notamment le cas de certaines applications de calcul scientifique et d'indexation de données. En revanche, lorsque le cahier des charges d'un SGF parallèle est plus stricte (cohérence forte, très haute disponibilité), il est beaucoup plus difficile d'obtenir des résultats satisfaisants, comme l'a montré l'exemple de xFS.

2.4.4 Systèmes de fichiers partagés

2.4.4.1 Introduction

Les systèmes de fichiers partagés (*shared file systems* ou *cluster file systems*) ont pris une importance croissante au cours des dix dernières années comme en témoigne l'augmentation de l'offre commerciale, qui s'est intensifiée depuis cinq ans. Un SGF partagé est basé sur l'exploitation d'un SAN (ou d'une couche logicielle fournissant une vue équivalente) permettant la communication directe entre chacun des nœuds d'une grappe et une matrice de disques. En contrepartie de ce pré-requis important, un SGF partagé répond à des besoins non satisfaits par les autres familles de systèmes.

- Il n'y a pas de point central par lequel transitent toutes les requêtes d'E/S. Les phénomènes de goulot d'étranglement et les problèmes de disponibilité sont ainsi amoindris.
- Un SGF partagé offre une sémantique de cohérence forte et des fonctionnalités de verrouillage avancées.
- Par rapport aux systèmes parallèles généralistes (tels que xFS), qui ne sont pas basés sur le paradigme de disque partagé, l'implémentation et l'administration du SGF sont plus simples. Les performances sont en outre nettement meilleures, en fonctionnement normal ainsi que lors d'une reprise après panne. Sur des grappes de moyenne échelle (quelques dizaines voire centaines de nœuds), ils exhibent généralement les meilleures performances d'E/S, toutes catégories de SGF confondues.
- Si la synchronisation des caches clients s'avère trop coûteuse ou si l'application gère elle même ces aspects, il est possible d'effectuer des « E/S directes » efficaces depuis les clients.

Les SGF partagés ont su trouver leur place sur un certain nombre de marchés pour lesquels un ou plusieurs des aspects mentionnés ci-dessus constituent des facteurs critiques :

- certaines applications scientifiques ayant des besoins de cohérence forte des caches clients et/ou de très hautes performances ;
- les serveurs de contenu multimédia (vidéo à la demande) ;
- les serveurs de courrier électronique et de news[66] ;
- les systèmes de gestion de base de données (voir 2.5).

Un système de fichiers partagé peut également être combiné à une à une approche clients-serveur de type NFS ou AFS (voir avec d'autres protocoles tels que FTP ou HTTP). Dans ce cas, un ensemble de nœuds centraux accèdent aux disques via un SGF partagé et réexportent l'arborescence de fichiers à une seconde catégorie de nœuds, moins privilégiés. Selon le(s) protocole(s) d'export employés entre les deux classes de nœuds, les garanties de cohérence fournies par le SGF partagé peuvent être perdues. De telles architectures sont cependant assez fréquentes (par exemple pour les serveurs de fichiers d'une entreprise), en raison des avantages qu'elles procurent en termes de tolérance aux pannes et de flexibilité d'administration.

On peut distinguer deux familles de SGF partagés : leur architecture étant *symétrique* ou *asymétrique*. Dans le premier cas, tous les nœuds sont égaux en termes de responsabilité et participent à la gestion des méta-données. Dans le second cas, la gestion des méta-données est confiée à un ou plusieurs serveurs particuliers mais les clients accèdent directement aux données. Cette section aborde ces deux modèles puis le cas de SGF partagés bénéficiant des fonctionnalités avancées d'un disque virtuel réparti sous-jacent.

[66] A titre d'exemple, voir [51, 124]. Ces retours sur expérience d'industriels insistent notamment sur les faiblesses des serveurs de courrier en grappe basés sur NFS ou sur des approches de partionnement/duplication (telles que [189]) et plaident en faveur des systèmes de fichiers partagés.

2.4.4.2 Systèmes de fichiers partagés « symétriques »

Cette partie présente trois exemples de systèmes partagés symétriques. Le premier est une référence historique, les deux suivants sont des produits commerciaux actuels.

VAXClusters Un des premiers SGF partagés symétriques fut intégré par Digital Equipment à sa plate-forme VAXclusters, il y a plus de vingt ans [127]. Celle-ci constitue elle même l'un des premiers exemples concluants de grappe de stations et de système à image unique. Un VAXcluster (VC) regroupe un ensemble de technologies propriétaires, matérielles (réseau à haut débit, contrôleurs réseau optimisés pour les communications inter-nœuds et le stockage de données) et logicielles (système d'exploitation VMS), développées conjointement.

Chacun des disques partagés peut être directement relié au réseau ou exporté par le nœud qui l'héberge ; le système de fichiers adresse de façon homogène ces deux topologies. Au niveau logiciel, le partage des ressources au sein de VMS est basé sur un gestionnaire de verrous. Dans le cas d'un VC, ce dernier est remplacé de manière transparente par un gestionnaire de verrous distribué (GVD ou DLM pour *distributed lock manager*) associé à un gestionnaire de grappe (cf 2.3.3) en charge du maintien d'une vue cohérente des autres nœuds. À l'instar des autres ressources partagées (services d'impression, d'ordonnancement de tâches, etc.), les accès aux fichiers sont synchronisés à l'aide du GVD.

Le GVD permet à une couche cliente d'acquérir ou de relâcher un verrou avec un certain mode (accès exclusif, lecture exclusive, etc.) et également de modifier le mode associé à un verrou (opération plus efficace que la création d'un nouveau verrou). Une requête de verrouillage est bloquante si elle est incompatible avec l'état courant de la ressource. Dans le même souci d'efficacité, le verrouillage peut être hiérarchique (une ressource peut être divisée en un arbre de sous-ressources et certaines modifications concurrentes peuvent être autorisées). La gestion des méta-données relatives au verrouillage est répartie sur l'ensemble (ou éventuellement un sous ensemble) des nœuds, sous la forme de deux services complémentaires : d'une part, la gestion des couplages entre ressources et gestionnaires, d'autre part, la gestion des verrous proprement dite. Lorsqu'un client (nœud A) émet une requête de verrouillage auprès de son GVD local, ce dernier détermine (via une fonction de hachage sur le nom de la ressource et l'ensemble des nœuds impliqués dans la gestion du verrouillage) quel nœud (B) contacter pour trouver le gestionnaire de la ressource et le contacte. Trois cas peuvent alors se présenter.

- Si B correspond au gestionnaire de la ressource, la requête est directement traitée et acquittée.
- Si B n'est pas le gestionnaire de la ressource mais que celle-ci a déjà été définie et confiée à un nœud C, B indique à A qu'il faut contacter C.
- Si B détermine que la ressource n'était pas définie au préalable, B indique à A qu'il devient le responsable de la ressource nouvellement définie.

Une requête de verrouillage nécessite ainsi deux messages dans le meilleur des cas et quatre dans le pire. Une requête de déverrouillage ne nécessite qu'un seul message. Dans tous les cas, le nombre de messages est indépendant de la taille de la grappe. Un numéro de version associé à chaque verrou permet, par ailleurs, de maintenir la cohérence des caches clients et un mécanisme de délégation permet à un client de déférer des écritures sur disque du moment qu'il ne retarde aucun autre client. L'ajout ou la suppression de nœuds à la grappe entraîne une suspension temporaire des services (tels que le SGF) pendant le temps nécessaire à la redistribution des rôles parmi les nœuds et à la réacquisition des verrous.

VAXclusters a constitué une plate-forme très impressionnante pour son époque en termes de performances, de robustesse et de simplicité d'administration. Cet exemple est relativement atypique car

les différents composants du système ont été développés et optimisés conjointement. Ceci n'est plus vraiment le cas aujourd'hui dans le contexte des grappes, ou tout du moins dans de bien moindres mesures. En particulier, l'intégration d'un SGF partagé au sein d'un noyau de système tel que Linux est loin d'être triviale pour les développeurs. L'architecture VAXclusters (et notamment son GVD) a cependant inspiré de nombreux SGF partagés par la suite.

Global File System (GFS) GFS est né à l'Université du Minnesota au milieu des années 1990 à partir d'investigations liées aux technologies émergentes pour SAN telles que Fibre Channel [206, 205, 176, 177, 178]. Il a ensuite été commercialisé par les sociétés Sistina puis Red Hat. Les travaux initiaux visaient principalement les applications scientifiques sous le système IRIX. Son usage a rapidement été étendu à un cadre plus généraliste (serveurs de fichiers, support des bases de données, etc) et au système Linux. Les caractéristiques de GFS ont assez largement évolué au fil des versions (six à ce jour) et nous ne retracerons pas toutes les étapes de ce cheminement. Sauf mention contraire, les descriptions se rapportent à la dernière version (et ne sont pas forcément applicables aux précédentes).

Comme évoqué dans l'introduction sur les SGF partagés symétriques, les données et méta-données sont stockées sur le même ensemble de disques partagés, et cogérées par l'ensemble des clients (il n'y a plus d'entité jouant le rôle de « serveur »). GFS utilise la notion de *groupe de ressources* (GR) pour disperser les structures de données décrivant l'état du système et favoriser le parallélisme d'accès. De façon schématique, un volume logique géré par le SGF est découpé en plusieurs mini-SGF (les GR) et chaque GR englobe de l'espace allouable pour les données mais également des zones réservées pour les méta-données du SGF. Dans le cas d'un SGF centralisé, cette technique permet d'augmenter les performances en laissant plusieurs contextes d'exécution manipuler simultanément différentes zones du SGF sans en compromettre l'intégrité. GFS étend ce principe à un cadre distribué, permettant à différents clients de modifier les méta-données de façon concurrente dans certains cas.

Les méta-données sont journalisées afin d'améliorer la fiabilité et d'accélérer la reprise sur panne. Chaque client doit maintenir un journal séparé et en cas de panne de l'un d'entre eux (détecté, comme d'habitude, via les services d'un gestionnaire de grappe), un « légataire » doit être désigné parmi les survivants pour remettre les méta-données en cohérence à l'aide du journal laissé par le nœud défunt.

Le superbloc associé par GFS à un volume logique a la particularité de contenir trois inodes spéciaux qui permettent d'étendre le système dynamiquement (ils permettent de stocker sous forme de fichier -donc modifiable- les principaux paramètres du système susceptibles d'évoluer au cours du temps).

- Le premier répertorie les différents GR associés à un volume logique et les informations associées (verrous, etc.).
- Le second décrit les différentes zones utilisées pour les journaux des différents clients.
- Le troisième recense les différents nœuds ayant monté le volume logique concerné et stocke pour chacun d'eux plusieurs informations : identifiant unique, nom de machine, adresse IP, identificateur du journal associé, méthode de *fencing* disponible (cf 2.3.3), identifiant du journal associé...

L'implémentation du SGF repose sur une couche de verrouillage générique qui permet d'utiliser, au choix, l'une des méthodes suivantes.

- Les premières implémentations tiraient partie d'une extension de la norme SCSI, les protocoles DLOCK/DMEP [23]. Un organe de stockage implémentant ces protocoles est doté d'une zone de mémoire permettant de stocker et manipuler de manière atomique un ensemble de données.

Cet espace peut être utilisé pour gérer des verrous. Cette solution permettait une gestion décentralisée et relativement efficace. Elle a cependant été abandonnée de manière progressive en raison de plusieurs obstacles, le principal étant le manque de matériel répondant à la norme. D'autres facteurs plus théoriques ont aussi motivé ce choix : le mécanisme est de très bas niveau et nécessite l'intervention du système (pour effectuer la correspondance entre un verrou « matériel » et l'entité logique à laquelle il est associé), son expressivité est limitée (impossible d'utiliser un mécanisme de callback), etc.

- Un serveur de verrouillage centralisé (s'exécutant de préférence sur un nœud réservé à cet effet) basé sur des canaux TCP/IP a également été développé pour pallier le manque de disques DMEP. Il ne s'agissait bien sûr que d'une solution de repli, contradictoire avec le modèle symétrique de GFS et ses objectifs.
- Enfin, la dernière version intègre un gestionnaire de verrous distribué à la VAXClusters.

GFS est aujourd'hui un système mature, fournissant des caractéristiques de stabilité et de performances satisfaisantes. Il présente en outre les avantages d'être disponible sous forme de logiciel libre et d'être accompagné d'un ensemble d'outils et de services (gestionnaire de grappe, gestionnaire de volumes logiques, etc.) pouvant être utilisés indépendamment et/ou bénéficier à d'autres couches d'un système en grappe.

General Parallel File System (GPFS) Il s'agit d'un SGF développé par IBM depuis une dizaine d'années. Initialement connu en tant que « Tiger Shark File System » consacré aux serveurs multimédia, il a été progressivement adapté à différentes architectures et types d'utilisation (avec une préoccupation toute particulière pour les très gros volumes de données organisés en très gros fichiers). Il équipe aujourd'hui l'ASCI White, l'un des supercalculateurs les plus puissants du monde (fournissant une capacité d'E/S de 7 Go/s) ainsi que de nombreuses grappes sous Linux [196].

GPFS a de nombreux points communs avec GFS. Il en diffère principalement par quelques optimisations et fonctions avancées. GPFS gère par exemple explicitement le striping et la duplication des informations sur plusieurs disques. Il bénéficie également d'heuristiques assez poussées permettant de détecter les schémas récurrents d'accès aux données (séquentiel, séquentiel inversé, etc.) ainsi que leur taille et d'ajuster en conséquence la politique d'allocation d'espace et de préchargement des caches. GPFS expose également une interface de programmation optionnelle permettant de paramétrer ces aspects au niveau applicatif.

Un aspect important de ce SGF concerne ses mécanismes hybrides de verrouillage. Il n'y a pas une méthode unique de maintien de la cohérence mais plusieurs, chacune étant spécialisée pour une opération précise en fonction du type d'information à synchroniser. Les trois principaux modes utilisés sont présentés ci-après.

- Un verrouillage par plage d'octets (*byte range locking*) est appliqué aux fichiers partagés. Lorsqu'un premier nœud accède à un fichier, un verrou intégral lui est attribué. Si un second nœud souhaite ensuite manipuler une autre portion du fichier, les deux nœuds peuvent négocier pour obtenir des verrous plus restreints. Ce dialogue direct entre clients permet de soulager le nœud en charge de la cohérence du fichier concerné.
- Pour le partage intensif de données (et/ou les accès concurrents à des données stockées sur un même bloc), il est possible d'utiliser le mode *data shipping* qui assigne un gestionnaire fixe (via un schéma de tourniquet) à chacun des blocs et évite de nombreuses communications distantes liées au GVD. En outre, la sémantique de cohérence peut être relâchée pour certaines applications scientifiques qui n'ont pas besoin de synchronisation à ce niveau.
- La stratégie de *metanode* est employée pour simplifier la synchronisation de certains scénarios d'accès concurrent aux méta-données d'un même fichier. Un verrou partagé sur un inode est

mise en œuvre en élisant un nœud responsable de fusionner les modifications périodiquement envoyées par les autres écrivains et de les écrire sur disque.

2.4.4.3 Systèmes de fichiers partagés « asymétriques »

Au delà de leur divergences architecturales avec les SGF symétriques, les SGF partagés asymétriques présentent généralement la particularité d'être hétérogènes, au sens où ils supportent des clients s'exécutant sur des environnements différents (à l'instar des systèmes clients-serveur tels que NFS). Le premier exemple illustre le concept de base d'asymétrie. Le second définit un modèle plus raffiné à partir des mêmes idées.

Clustered XFS (CXFS) CXFS [77, 203] est une extension du SGF local XFS, développé par la compagnie SGI et présentant des fonctionnalités avancées de journalisation. Il est commercialisé depuis environ cinq ans.

La gestion des méta-données est confiée à un serveur central (il est possible d'utiliser plusieurs serveurs, chacun étant responsable d'un volume logique monté par les clients). Les clients communiquent avec le serveur via des canaux TCP/IP, sur un réseau différent de celui du SAN, et de préférence consacré uniquement à cet usage. Il est souhaitable que ce réseau fournisse de bonnes performances en latence mais le débit n'est pas important, contrairement à celui du SAN, car les requêtes sur les méta-données sont de petite taille. Celles-ci sont divisées en plusieurs catégories (date d'édition, taille, localisation des données, table d'allocation des blocs, etc.) et chacune d'elle est associée à un jeton, autorisant la mise en cache d'informations en mode lecture ou écriture. La perte d'un jeton indique à un client qu'il doit synchroniser les informations concernées avec le serveur. Cette granularité assez fine de synchronisation permet de limiter les communications avec le serveur (ainsi que la taille des messages associés). La gestion des verrous est également confiée au serveur central.

Comme dans le cas des SGF parallèles basés sur une topologie relativement proche, le serveur de méta-données n'est, dans les faits, généralement pas un goulot d'étranglement. Il est cependant un point très sensible aux défaillances mais la présence des disques partagés simplifie la reprise après une panne (détectée par les méthodes habituelles). Si le serveur ne redémarre pas, un autre nœud peut être chargé de cette tâche et récupérer les informations nécessaires sur les disques.

Storage Tank Ce produit récent (2003-2004) d'IBM est relativement ambitieux. Storage Tank[67] (ST) [145] est proposé comme épine dorsale pour l'infrastructure de stockage d'une entreprise, et a par conséquent une vocation généraliste. Il n'a, à notre connaissance, pas encore été largement diffusé et nous ne disposons pas de résultats expérimentaux sur ses performances. Nous en présentons néanmoins les caractéristiques principales.

Les principaux objectifs de ST concernent le passage à l'échelle (gérer des milliards de fichiers et de milliers voire millions de clients) et la réduction des coûts humains en matière d'administration (semi-automatisation des tâches fréquentes et de décisions relatives au placement des données et à l'optimisation de l'usage des ressources - *policy-based management*).

Le stockage des données est organisé autour de deux concepts principaux : conteneurs et groupes de ressources (*storage pools*). Un conteneur est associé à une sous arborescence de l'espace de noms global. Cette unité de structuration logique (dimensionnée pour environ cent millions de fichiers) est utilisée pour la répartition de charge, la sauvegarde de données ou la reprise sur panne. Un groupe de ressources (GR) fédère un ensemble de volumes et alloue de l'espace de stockage aux conteneurs. Un GR est associé à des attributs (performance, localisation, ...) destinés à aider les choix d'allocation

[67] Egalement connu sous le nom de « SANFS ».

d'un conteneur. Un conteneur peut utiliser l'espace de plusieurs GR et la capacité d'un GR peut être partagée entre plusieurs conteneurs. Ces abstractions servent de briques de bases à un ensemble de fonctionnalités dynamiques de sauvegarde, *snapshot*, migration de données et répartition de charge.

L'architecture de ST, prévue pour une grande échelle (en termes de nombre de nœuds mais aussi, dans une certaine mesure, en termes de distance) fait moins d'hypothèses que les autres SGF partagés quant au couplage entre les différents nœuds. En particulier, chacun des clients n'a pas à connaître ses pairs (ce qui est habituellement le cas, y compris pour CXFS) et il n'y a pas besoin d'utiliser un gestionnaire de grappe pour détecter et réagir à la panne d'un client. Le seul interlocuteur d'un client (à l'exception des organes de stockage, auxquels il a directement accès) est le serveur de méta-données. Ce dernier peut détecter la panne d'un client via l'expiration d'un bail et a la responsabilité de la reprise après panne (assez simple). Le serveur peut en outre être réparti sur plusieurs nœuds pour les motivations classiques de tolérance aux pannes et de passage à l'échelle (préoccupation importante vu le nombre de clients et de ressources considérés). La responsabilité des méta-données est répartie dynamiquement entre les serveurs (avec une granularité de niveau conteneur), ce qui permet d'harmoniser la charge et de réagir simplement à une évolution du nombre de serveurs (volontaire ou accidentelle). Ainsi, seul le groupe de serveurs (restreint par rapport à l'échelle du système) nécessite un couplage fort. Par ailleurs, le serveur de méta-données est implémenté au niveau utilisateur, de manière portable et harmonise les sémantiques des différents systèmes d'exploitation supportés. ST peut ainsi être déployé sur des architectures hétérogènes, aussi bien pour les clients que pour les serveurs.

Un client communique avec un serveur par l'intermédiaire d'un protocole optimisé (*ST protocol*) développé au dessus d'IP, qui prend en charge les nécessités de verrouillage et de cohérence des caches, la réaction aux pannes de serveurs, ainsi que des fonctionnalités avancées (*copy-on-write* pour les *snapshots*, sauvegardes). Le mécanisme de verrouillage découple les requêtes clientes d'ouverture et de fermeture de fichier et le maintien de la cohérence des caches. Ceci permet des stratégies très poussées de gestion de cache sans compromettre l'intégrité des données. La portion cliente du SGF peut par exemple précharger un fichier sur un nœud avant qu'une application ne l'ait ouvert. Contrairement à GPFS, la négociation directe de verrous entre clients est impossible (en raison du couplage faible imposé) mais un serveur peut agir comme médiateur et dégrader la portée d'un verrou pour permettre l'accès concurrent à un fichier lorsque les modes de manipulation sont compatibles.

Les projets de développement autour de ST concernent principalement la gestion des organes de stockage à objets, la collaboration entre plusieurs SAN distants (migration dynamique de données, etc.) et la poursuite de l'automatisation des tâches d'administration.

2.4.4.4 Systèmes de fichiers partagés basés sur un disque virtuel réparti

Outre l'émulation de disques partagés onéreux, certains disques virtuels répartis ont également pour objectif de simplifier le développement de systèmes de fichiers répartis en fournissant plus de services à cette couche qu'une interface « classique » de niveau blocs (telle que celle exportée par l'implémentation matérielle d'un SAN). Nous présentons ici quelques exemples représentatifs des avantages qu'un DVR peut procurer à la couche de niveau supérieur qui l'exploite.

Frangipani/Petal Le DVR Petal (évoqué en 2.3.2.2) a été employé comme substrat de base pour le SGF Frangipani [222]. Le couple Petal/Frangipani fonctionne de manière assez similaire aux SGF partagés que nous avons présentés. Les trois principales entités du système sont le module SGF Frangipani (qui s'appuie sur le pilote/talon client Petal), le serveur de disque virtuel Petal et le gestionnaire

de verrous. Ces trois entités peuvent cohabiter sur chacun des nœuds de la grappe (cas classique)[68] mais cela n'est pas obligatoire. Chaque client utilise un journal (comme dans l'exemple de GFS) pour simplifier la reprise sur panne. Le maintien de la cohérence des données est assuré par l'ensemble des gestionnaires coopératifs de verrous. Ceux-ci sont synchronisés grâces aux mêmes mécanismes que ceux employés par Petal pour les informations de couplage du disque virtuel. Un verrou (d'une granularité assez grosse, un fichier complet) est associé à un bail, dont l'expiration permet de détecter la panne d'un client. Les principaux apports de Petal envers Frangipani sont décrits ci-dessous.

- Le concept de volume virtuel permet de simplifier le placement des structures de données (qui englobent les méta-données) sur disque. De la même façon que pour un espace mémoire virtuel, différentes zones peuvent être définies et l'espace de stockage n'est alloué que pour les emplacements dans lesquels des informations ont été effectivement écrites. Ainsi, quelle que soit la puissance de stockage de l'infrastructure physique sous-jacente, le SGF manipule une partition virtuelle de taille fixe (16 exaoctets soit 2^{64} octets). En contrepartie, l'abstraction des ressources de stockage empêche intrinsèquement toute optimisation de placement des données par le SGF (ce que font certains SGF tels que GPFS)[69].
- Le code du SGF proprement dit est simple car chacun des clients fonctionne indépendamment des autres. Les seules parties du système fortement couplées sont les serveurs de stockage (Petal) et ceux de verrouillage. Ainsi, des clients peuvent être ajoutés ou supprimés sans perturber le fonctionnement des autres nœuds ayant monté le SGF.
- Le SGF bénéficie des fonctionnalités de *snapshot* fournies par le disque virtuel[70].

Malgré son potentiel prometteur et le fait qu'il ait été développé dans un contexte industriel, le prototype Petal/Frangipani n'a curieusement pas donné naissance à des produits commerciaux.

Shared Logical Disk (SLD) Des travaux assez proches de Petal ont également été menés à l'université de Princeton à la même époque (1995-97) et ont abouti à des conclusions similaires [201]. Une différence significative entre Petal et SLD est que ce dernier (à l'instar de RAID-X) gère la cohérence des caches clients. Ceci permet de simplifier la mise en œuvre d'un SGF réparti, qui, dans ces conditions peut être dérivée d'une implémentation centralisée si trois contraintes sont respectées.

- Le SGF doit verrouiller chaque bloc avant de l'utiliser.
- Le SGF ne doit pas conserver des informations (données et méta-données) en cache.
- Afin de pouvoir supporter des écritures retardées et tolérer la panne d'un client sans interruption de service, le SGF doit spécifier un ordre d'écriture pour certaines séquences de mise à jour des méta-données[71]. Pour cela, SLD fournit une primitive (non standard) au SGF, qui permet d'indiquer une contrainte d'ordre sur l'écriture de deux blocs (de méta-données).

Cette approche présente toutefois son lot de défauts. D'une part, la cohérence des caches est imposée au niveau blocs (et en conséquence, les couches supérieures ne sont pas autorisées à gérer leurs propres caches), ce qui exclut d'employer un tel DVR avec des applications prenant elles-mêmes en charge leur cache de blocs et les problèmes associés. D'autre part, rapprocher (voire remplacer) un

[68] Les tests de performance n'ont cependant considéré que des topologies répartissant clients Frangipani et serveurs Petal sur des nœuds distincts.

[69] Ceci n'est pas un handicap insurmontable. Des heuristiques de placement de données peuvent être développées au niveau du disque virtuel réparti.

[70] Toutefois, pour obtenir un *snapshot* cohérent au niveau du SGF sans recourir à l'examen du journal de chaque client, une synchronisation globale de tous les clients (via le service de verrouillage) est nécessaire.

[71] Cet aspect est passé sous silence dans la description de RAID-X, qui a été utilisé conjointement avec Ext2, un SGF local pour Linux.

SGF partagé d'un SGF local est louable dans l'optique des systèmes à image unique mais les solutions proposées jusqu'à présent ignorent certains problèmes de reprise sur pannes[72].

Pour conclure sur ce point, il convient de mettre en relation Frangipani et xFS. Ces deux SGF ont été développés à la même époque, à partir d'hypothèses matérielles similaires et avec des objectifs semblables (aucun point central, cohérence forte, reconfiguration dynamique). Le fait de reposer sur l'abstraction de disque partagé a permis pour Frangipani une mise en œuvre beaucoup plus simple que celle de xFS, et des performances significativement plus élevées. Ceci démontre en outre que les multiples niveaux d'indirection de l'architecture Petal/Frangipani n'ont pas un surcoût indésirable par rapport aux avantages qu'ils procurent (reconfiguration transparente, *snapshot*).

2.4.4.5 Bilan sur les systèmes de fichiers partagés

Les SGF partagés offrent des caractéristiques remarquables en termes de mécanismes de verrouillage, sémantique de cohérence forte, haute disponibilité, reconfiguration dynamique et performances brutes d'accès aux données. En contrepartie, ils s'avèrent complexes à implémenter et reposent sur un pré-requis important : chacun des clients doit pouvoir accéder directement à l'ensemble des organes de stockage. Ce pré-requis peut être satisfait de manière matérielle ou logicielle. La première solution nécessite le déploiement d'une infrastructure SAN au coût d'acquisition élevé. La seconde solution repose sur l'emploi d'un disque virtuel réparti. Cette approche est généralement moins efficace que la première mais peut présenter des caractéristiques intéressantes (en plus de son aspect économique) : un disque virtuel réparti n'est pas limité à la « simple » émulation d'un SAN et peut offrir des services complémentaires qui simplifient l'implémentation et le fonctionnement des SGF partagés : snapshot, clôture d'E/S (*I/O fencing*) logicielle pour refuser les écritures émanant d'un client jugé défectueux, etc.

2.4.5 SGF répartis pour interfaces de stockage à objets

Les principes de base et l'intérêt du stockage à objets (OSD) ont déjà été présentés en 2.1.2.5. Les travaux fondateurs concernant cette nouvelle interface (menés par le projet NASD -*Network Attached Secure Disks*- à Carnegie Mellon University au milieu des années 1990 [83, 84]) ont d'emblée été appliqués à un contexte réparti et ont ouvert la voie aux systèmes de fichiers partagés asymétriques (décrits en 2.4.4.3). L'architecture NASD présente des particularités en termes de décentralisation et de sécurité.

- En plus de présenter une interface à objets plutôt qu'à blocs, les disques utilisés sont équipés d'une interface réseau. Ainsi une matrice de disques NASD n'a pas nécessairement de contrôleur central. Il en résulte une architecture plus décentralisée : le mécanisme de répartition/redondance des données peut être implémenté au niveau de chaque client. Le système de stockage n'est ainsi pas contraint par les performances et la robustesse d'un contrôleur centralisé pour SAN. En contrepartie, les schémas de redondance par parité (tels RAID 5) sont beaucoup plus complexes à implémenter car ils nécessitent des protocoles de synchronisation spécifiques (cf 2.3.2) [7, 6].
- Contrairement aux SGF partagés que nous avons présentés précédemment, les contraintes de sécurité retenues étaient plus strictes et une partie des travaux sur les disques intelligents ont porté sur la définition d'un protocole sécurisé pour vérifier les droits d'accès d'un client vis-à-vis des objets stockés sur disque. Ce protocole est basé sur le partage d'un secret entre le

[72] Par exemple, ne gérer qu'un seul journal pour tous les clients est complexe et peu efficace. Ne pas en gérer est contraire aux objectifs de disponibilité d'un SGF partagé.

serveur et chacun des disques OSD. Lorsqu'un client contacte un serveur de méta-données pour accéder à un fichier, le serveur lui attribue une capacité qui permet au disque OSD de vérifier si l'accès aux données doit être accepté.

Il n'existe pas encore de produits commerciaux implémentant de façon matérielle les caractéristiques attendues d'un disque OSD. Toutefois, les efforts de standardisation de l'interface OSD (sous la forme d'un nouveau jeu de commandes SCSI) progressent rapidement au niveau d'instances telles que le SNIA et l'ANSI T10, si bien que les premiers véritables périphériques du genre devraient apparaître d'ici cinq ans. En parallèle, les travaux de recherche se poursuivent notamment pour l'amélioration des protocoles de sécurité (en termes de robustesse mais également de coût en ressources d'exécution, en vue de simplifier leur embarquement au sein des disques) ainsi que pour l'intégration d'interfaces de communication réseau.

En attendant, les différents systèmes OSD disponibles actuellement reposent sur une implémentation logicielle des concepts de stockage à objets. Ainsi, les périphériques OSD déployés actuellement ne sont « rien d'autre » que des serveurs affectés uniquement à la tâche des E/S, exportant un nouveau type d'interface et stockant *in fine* les objets dans les fichiers d'un SGF local. C'est notamment le cas des matrices de stockage à objets pour lesquelles ont été adaptés les SGF Slice et Storage Tank (cf 2.4.2.4 et 2.4.4.3). C'est aussi le cas de celles employées par de nouveaux SGF, nativement conçus pour l'interface OSD tels que Lustre[73] [54, 31] et ActiveScale[74] [168]. Ces deux derniers SGF (relativement similaires entre eux) sont assez proches d'un système asymétrique tel que Storage Tank. Par conséquent, nous ne détaillerons pas leur fonctionnement global mais uniquement quelques exemples d'optimisations[75] proposées par Lustre.

- Les interactions nécessaires entre un client et un serveur de métadonnées sont limitées grâce à l'utilisation d'opérations groupées. Lorsqu'un client veut créer un fichier, il doit habituellement commencer par demander un verrou sur le répertoire ciblé puis demander la création du fichier. Dans le cas de Lustre, le client profite de sa demande de verrou pour indiquer qu'il souhaite créer un fichier. Si la requête peut être satisfaite par le serveur, alors ce dernier crée directement un fichier et retourne un verrou correspondant. Par ailleurs, certaines communications entre un serveur de méta-données et un nœud de stockage peuvent également être évitées car le serveur conserve un cache d'objets préalloués pour accélérer les réponses aux clients demandant la création d'un fichier.
- Le code du SGF est basé sur une couche d'abstraction pour les communications [194] permettant d'utiliser de nombreux protocoles/technologies réseau (éventuellement capables de RDMA, d'E/S directes, etc...) de façon transparente . En outre, différents clients peuvent utiliser différents protocoles de communication, dont l'intéropérabilité est assurée par des modules de « routage ».

Lustre et ActiveScale sont des SGF généralistes et sont prévus pour faire face à des charges d'E/S avec écritures concurrentes. D'autres SGF répartis à objets sont plus spécialisées. Le projet Sorrento [220, 219] vise par exemple les applications à faible partage en écriture tels que les moteurs d'indexation de documents ou les programmes de traitement d'images médicales, en utilisant les disques locaux des nœuds d'une grappe. Il rejoint en cela la vocation de GoogleFS (cf 2.4.3.2) mais, contrairement à ce dernier, supporte toutes les opérations d'un SGF classique, offre une sémantique de

[73] logiciel libre développé par la société Cluster File Systems et financé principalement par l'*US Department of Energy*
[74] produit commercial de Panasas, société dérivée du projet NASD
[75] Remarque : ces optimisations ne sont pas spécifiques à un modèle de stockage à objets.

cohérence forte et manipule efficacement les petits fichiers. Sorrento est basé sur un modèle asymétrique (serveurs de noms et nœuds de stockage[76]) et un découpage en segments pour chaque fichier. Cette structure des fichiers est assez proche de celle définie par GoogleFS mais elle est mieux exploitée par Sorrento car elle se prête bien à un couplage avec la notion d'objets. Par ailleurs, Sorrento emploie des techniques inspirées des systèmes *pair-à-pair* (algorithmes de hachage) pour distribuer la gestion des méta-données. Les serveurs de noms ne stockent pas d'information sur l'emplacement des différents objets (segments) qui composent un fichier car un postulat de ce système est que de nombreux nœuds peuvent rejoindre ou quitter la grappe à tout moment. Des techniques optimisées pour les cas fréquents (contrôle optimiste des modifications concurrentes via des numéros de version associés aux segments, *copy-on-write*, synchronisation différée des réplicas) permettent l'obtention de bonnes performances tout en maintenant une cohérence forte des données. Le support d'applications spécialisées travaillant sur des ensembles de données disjoints, aujourd'hui assuré par des SGF tels que PVFS et GoogleFS évoluera sans doute vers des architectures à la Sorrento, qui fournissent des performances comparables et plus de capacités d'auto-adaptation aux variations structurelles de la grappe.

Du fait de l'émulation logicielle de la couche OSD des périphériques, les SGF répartis à objets d'aujourd'hui ne fournissent pas encore leur plein potentiel en termes de performances. Néanmoins, les résultats actuels sont déjà très encourageants. Lustre, bien qu'encore en cours de développement, est d'ailleurs déjà déployé avec succès sur plusieurs grappes du top 10 mondial (1000 nœuds ou plus). La topologie de *SGF partagé asymétrique* est probablement amenée à dominer le panorama des très grandes grappes de calcul (voire d'autres secteurs) dans les années qui viennent.

2.5 Systèmes de gestion de bases de données

La problématique de l'exploitation en grappe de bases de données est relativement proche de celle des SGF répartis. Une des principales différences par rapport au contexte des SGF tient au fait que la très grande majorité des applications nécessitent impérativement une cohérence forte des données. On peut distinguer trois topologies d'exploitation de bases de données en grappes : serveur central, duplication/partitionnement et disques partagés.

Une approche centralisée est basée sur un modèle client-serveur avec les avantages (simplicité d'administration) et les inconvénients (goulot d'étranglement et point central de défaillances) associés. De plus, le goulot d'étranglement est accentué par le fait qu'un SGBD consomme en général plus de ressources qu'un SGF. En conséquence, le serveur doit disposer d'une configuration appropriée (machine multi-processeur très onéreuse).

La seconde topologie permet plus de parallélisme et une meilleure tolérance aux pannes en utilisant plusieurs copies de certaines tables réparties sur différents nœuds. Un tel dispositif peut être intégré au sein d'un SGBD (c'est par exemple le cas dans DB2 [109] développé par IBM) ou implémenté par un intergiciel (tel que C-JDBC [45]) déployé au dessus de plusieurs SGBD centralisés (et potentiellement hétérogènes) installés sur différents nœuds. Cette approche est bien adaptée aux applications qui effectuent principalement des accès en lecture, telles que les sites web de commerce électronique (pour lesquels environ 85% des requêtes vers la base de données correspondent à des lectures).

La dernière méthode est basée sur l'emploi d'un SAN et de techniques relativement analogues à celles d'un SGF partagé. Elle est utilisée dans le cadre des applications intensives en écriture (par

[76] Ces deux types d'entités logicielles ainsi que les clients peuvent cohabiter sur un même nœud.

exemple la fouille de données) et/ou avec d'importantes contraintes de performances et de haute disponibilité. Un SGBD pour SAN tel qu'Oracle RAC [163] utilise généralement des techniques de cache coopératif pour limiter les lectures sur disque après la mise à jour d'une information (il y a, de manière générale, significativement plus d'accès concurrents aux mêmes données ou métadonnées dans le contexte des SGBD que dans celui des SGF, ce qui limite l'intérêt d'un « simple » cache local). Pour fournir des résultats probants, une telle stratégie (gestionnaire de verrous distribué combiné avec un cache coopératif) nécessite l'utilisation d'un réseau à hautes performances (le critère primordial étant une faible latence). Par ailleurs, précisons qu'il n'est pas incompatible d'utiliser un SGBD parrallèle pour grappe avec un SGF partagé (tel que GFS ou GPFS). Une telle architecture permet de faciliter l'administration du SGBD (notamment en mutualisant de nombreux fichiers de configuration entre tous les nœuds de la grappe et en permettant une gestion des volumes plus flexible) sans sacrifier les performances car le SGBD peut accéder par *E/S directes* aux volumes utilisés pour stocker les données.

2.6 Perspectives

Cette section présente les perspectives à court et moyen terme pour les systèmes de stockage répartis à l'échelle des grappes de machines. Deux catégories de recherches sont distinguées : les modifications architecturales appliquées aux matrices de disques et les systèmes de stockage semi-autonomes.

2.6.1 Evolution des matrices de disques (briques de stockage)

2.6.1.1 Orientations générales

Au delà des modifications d'interface (du bloc à l'objet) que nous avons déjà mentionnées, les industriels sont également en train d'envisager des modifications architecturales profondes pour la conception des périphériques de stockage en masse. Ces recherches sont motivées par les limitations actuelles des matrices de disques. Les contraintes (électroniques) d'intégration et d'optimisation sont telles que les modèles courants sont très peu extensibles. Il en résulte des coûts très élevés de développement (et a fortiori d'acquisition) ainsi qu'un manque d'évolutivité.

Pour résoudre ce problème, une approche émergente consiste à remplacer les matrices traditionnelles et leur contrôleur centralisé par un assemblage de « briques intelligentes » [107, 75, 190]. Une brique est un serveur optimisé intégrant un processeur rapide, de la mémoire NVRAM (pour la mise en cache de données et simplifier la reprise sur panne), un ensemble de disques (pour un total de l'ordre de quelques téraoctets) et une ou plusieurs interfaces réseau à haut débit (pour relier les briques aux clients du SAN ainsi que les briques entre elles). Les architectures envisagées regroupent jusqu'à plusieurs milliers de briques. La redondance de données (pour la tolérance aux fautes et le parallélisme des requêtes) est gérée à deux niveaux :

- (éventuellement) à l'échelle d'une brique, par les techniques traditionnelles de RAID ;
- (obligatoirement) à l'échelle du « cube » (l'assemblage de briques), par duplication de segments sur plusieurs briques (par exemple choisies aléatoirement parmi l'ensemble des briques non pleines).

Par rapport au contrôleur central d'une matrice de disques classique, celui d'un cube est nettement simplifié. Soit il n'a plus qu'une fonction de routage de requêtes (distribution de charge, sans gestion de la synchronisation des données dupliquées), soit il est carrément inexistant (et la fonction de routage est déportée dans le pilote de chaque client). Dans le second cas, chaque client de ce nouveau

type de SAN peut contacter indifféremment n'importe quelle brique. Ces modifications impliquent de nouveaux défis, dont ceux listés ci-après.

Synchronisation efficace des briques Afin de préserver l'intégrité des données dupliquées, les différentes briques doivent se coordonner pour garantir un ordre global sur les écritures des données ainsi que les cohérence des opérations de reconfiguration (ajout ou disparition d'un nœud ou d'un disque) [76, 190].

Distribution optimale de la charge Pour que le cube fournisse des bonnes performances, les requêtes d'E/S doivent être réparties convenablement entre les briques, qui sont potentiellement hétérogènes[77]. Dans ce but, il est nécessaire de prendre en compte des facteurs tels que l'utilisation des différents caches ainsi que la position des têtes de lecture des différents disques, ce qui est plus complexe dans un contexte distribué [138].

2.6.1.2 Un exemple : *Federated Array of Bricks*

L'une des premières infrastructures logicielles d'exploitation de briques de stockage est FAB (*Federated Array of Bricks*) [75, 190], développée par HP Labs. A l'instar de Petal, FAB repose sur l'emploi d'une table de correspondance à plusieurs niveaux pour gérer les métadonnées d'un disque virtuel réparti. Les informations nécessaires globalement sont dupliquées sur toutes les briques et les mises à jour sont effectuées de manière cohérente grâce à un protocole de consensus (tel que Paxos) permettant d'obtenir un *broadcast* atomique.

Une différence majeure entre Petal et FAB concerne la façon dont est gérée la duplication des données. Petal repose sur un protocole de duplication maître-esclave, source de deux problèmes principaux : il n'est pas tolérant aux partitions de réseau et peut également causer, lors d'une reprise sur panne, une période d'indisponibilité qui peut être perceptible par les clients et les inciter à démarrer à leur niveau un mécanisme coûteux de gestion de fautes. Par opposition, FAB peut masquer les pannes de nœuds ou de lien réseau de façon transparente et rapide grâce à une méthode de vote. Lorsque une brique reçoit une requête de la part d'un client, celle-ci devient la coordinatrice de cette requête : pour une écriture, elle génère une nouvelle estampille (globalement unique) et s'assure qu'une majorité de briques a reçu la requête ; pour une lecture, elle récupère l'estampille (associée au bloc concerné) depuis une majorité de briques et retourne les données de la version la plus récente. Un ensemble de protocoles optimisés (sans besoin de maintien d'un état persistant sur le coordinateur) permet de garantir un ordre global sur la mise à jour des blocs de données.

Ces protocoles sont compatibles avec une duplication complète des données (plusieurs exemplaires complets d'un même bloc de données sont conservés sur différents nœuds) mais également avec des schémas de redondance à la Reed-Solomon, moins efficaces mais moins gourmands en place. Il est ainsi possible d'envisager un large éventail de configurations, plus ou moins efficaces, économiques et robustes, en fonction du schéma de redondance, du nombre de briques et du taux de duplication choisis. Un avantage supplémentaire de cette stratégie basée sur le principe de vote tient au fait qu'elle est très bien adaptée aux environnements hétérogènes (cas des très grands cubes dont la capacité est augmentée de manière incrémentale) car elle masque les mauvaises performances des briques les plus lentes. Par ailleurs, un protocole de reconfiguration permet de modifier dynamiquement la structure d'un disque virtuel réparti (ajout ou suppression de nœuds ou de disques). Ce dernier englobe une phase de consensus sur le changement de la « vue du système », préalable à une phase de synchronisation d'état.

Grâce à ces deux mécanismes (vote sur l'estampille des données et structure dynamique de quorum), le système de stockage peut fonctionner de manière totalement décentralisée et ainsi garantir

[77] Comme l'extensibilité est l'un des objectifs majeurs de cette nouvelle architecture, un cube peut regrouper plusieurs générations de briques, aux capacités (de traitement et de stockage) assez différentes.

une très bonne disponibilité. Les clients communiquent avec les briques via le protocole iSCSI pour les requêtes de données et via un protocole propriétaire pour les tâches d'administration (création de volumes logiques, ...). La répartition de charge et la tolérance aux pannes peuvent être gérées directement par le pilote client (paramètré avec une liste de briques disponibles) ou de façon transparente par l'intermédiaire d'un service de désignation (tel qu'iSNS).

Le prototype FAB, implémenté en mode utilisateur à été déployé sur une vingtaine de briques. Les expériences ont montré le passage à l'échelle de système par ajout de briques, les bonnes performances de la stratégie de vote par rapport à l'approche plus classique de duplication maître-esclave et une bonne réactivité aux pannes, dont la plupart sont complètement masquées aux clients.

En complément des considérations liées aux économies de développement, un second facteur ayant favorisé le développement des briques intelligentes est la volonté d'accroître l'autonomie des systèmes de stockage. Cet axe de recherche est détaillé dans la section suivante.

2.6.2 Systèmes de stockage autonomes

Ces dernières années ont vu l'essor d'un nouveau mot d'ordre au sein de l'industrie informatique (avec des acteurs tels qu'IBM en fer de lance) : les systèmes autonomes (*Autonomic Computing*) [118, 108]. L'objectif est de remédier à la complexité croissante des machines en construisant des systèmes informatiques capables de s'auto-gérer (selon quatre axes principaux : configuration, optimisation, protection, guérison), ne laissant aux administrateurs humains que la tâche d'exprimer les « macro objectifs » attendus de leur comportement. Ces efforts de recherche, qui empruntent des concepts et techniques développés dans des domaines tels que l'intelligence artificielle et l'automatique, s'intéressent notamment au contexte des grappes de machines [14] ainsi qu'à celui des systèmes de stockage. Il est en effet notoire que cette dernière tâche nécessite de très importantes ressources humaines. Certaines études estiment même qu'un administrateur à plein temps est nécessaire pour 10 téraoctets de données (voire moins), perspective assez effrayante alors que des infrastructures à l'échelle du pétaoctet seront bientôt courantes.

Le projet « Self-* »[78] de Carnegie Mellon University est l'un des pionniers dans le domaine du stockage autonome [78]. Les principaux objectifs attendus de cette nouvelle génération de systèmes sont présentés ci-dessous, selon cinq axes principaux.

Pérennité et intégrité des données Le système doit automatiquement organiser et maintenir la redondance des données, ainsi qu'effectuer régulièrement des *snapshots* et des sauvegardes (locaux et distants) pour permettre la restauration d'un état antérieur des informations.

Optimisation des performances Le système doit être capable de s'auto-optimiser en fonction des charges d'E/S qui lui sont couramment injectées, en jouant sur les paramètres des différentes couches logicielles impliquées, les politiques de répartition de charge, la migration de données, etc. Les administrateurs ne sont sollicités que pour donner des indications à gros grains sur les résultats insatisfaisants, les garanties à fournir (taux de probabilité acceptable pour les pannes causant une interruption de service, temps de reprise sur panne), les compromis à adopter (par exemple, privilégier les performances au détriment du volume physique occupé).

Planification et déploiement Le système doit pouvoir intégrer automatiquement de nouvelles ressources à un parc matériel en cours d'utilisation. Il doit également jouer un rôle d'aide à la

[78] Le nom du projet désigne des systèmes capables de tout faire par eux-même, en référence au caractère joker « * » utilisé dans de nombreux programmes.

décision pour les administrateurs en leur indiquant à quel moment il est nécessaire d'acquérir de nouvelles ressources.

Supervision Le système est en charge de s'auto-surveiller et de conserver des informations détaillées (et intelligibles) sur sa structure interne (et l'historique de son évolution).

Diagnostics et réparations Un système doit pouvoir réagir par lui même à la plupart des défaillances en isolant le problème et en effectuant les reconfigurations adéquates. Un administrateur ne doit être réquisitionné que pour effectuer le remplacement d'équipements défectueux grâce aux indications précises du système.

L'architecture proposée par le projet Self-*, inspirée par la structure classique d'une entreprise, est assez différente des schémas actuellement déployés. Celle-ci, représentée en figure 2.5 est basée sur une hiérarchie de gestionnaires/superviseurs qui contrôlent et ajustent le fonctionnement d'ouvriers (des briques de stockage intelligentes) auxquels des clients externes accèdent par l'intermédiaire de routeurs. Il s'agit d'entités logiques, dont plusieurs types peuvent cohabiter au sein d'un même ressource physique.

- Un superviseur a pour mission de répartir la gestion des données et des objectifs entre ses différents subordonnés (superviseurs de plus bas niveau ou ouvriers) et de vérifier que le sous-arbre hiérarchique dont il est responsable fournit les résultats attendus. Les décisions prises par un superviseur sont basées sur trois principaux indicateurs remontés par les subordonnées : un profil de la charge d'E/S associée au différents types de données stockées, des informations sur les capacités de chacun des ouvriers et des prédictions sur l'évolution de ces dernières en fonction des variations de charge. De façon complémentaire à la hiérarchie de décision, un certain nombre d'*assistants* fournissent des services transversaux (annuaire, journaux d'événements, authentification ...). Enfin, le superviseur principal sert d'interlocuteur avec l'administrateur humain.
- Un routeur (analogue à la notion de « contrôleur de cube » vue en 2.6.1) achemine les requêtes de clients externes[79] vers l'(es) ouvrier(s) approprié(s) (en fonction des données concernées et de l'état courant du système, dont sa charge). La fonction de routeur peut être intégrée au pilote du client, intégrée au sein de chaque ouvrier, ou encore placée sur un nœud intermédiaire entre les clients et les ouvriers.
- Les ouvriers ont pour rôle de répondre aux requêtes d'accès aux données qu'ils stockent sur disque. Ils effectuent également les tâches d'administration (migration de données, sauvegardes, etc.) ordonnées par les superviseurs (les mouvements de données sont directs, de brique à brique), à qui ils fournissent les indicateurs mentionnés ci-dessus.

Si les premiers prototypes de briques intelligentes sont déjà bien avancés chez les constructeurs, les recherches sur les macro-systèmes autonomes tels que celui envisagé par Self-* ne sont encore qu'embryonnaires, et beaucoup de chemin reste à parcourir avant que l'administration quasi-automatique d'un parc de stockage devienne une réalité.

[79] Par opposition aux *clients internes*, considérés comme partie intégrante du système. Un client interne est par exemple un serveur NFS ou un SGBD qui stocke des données sur les briques et les exporte vers des clients non modifiés. Un client interne peut typiquement être co-localisé avec un routeur.

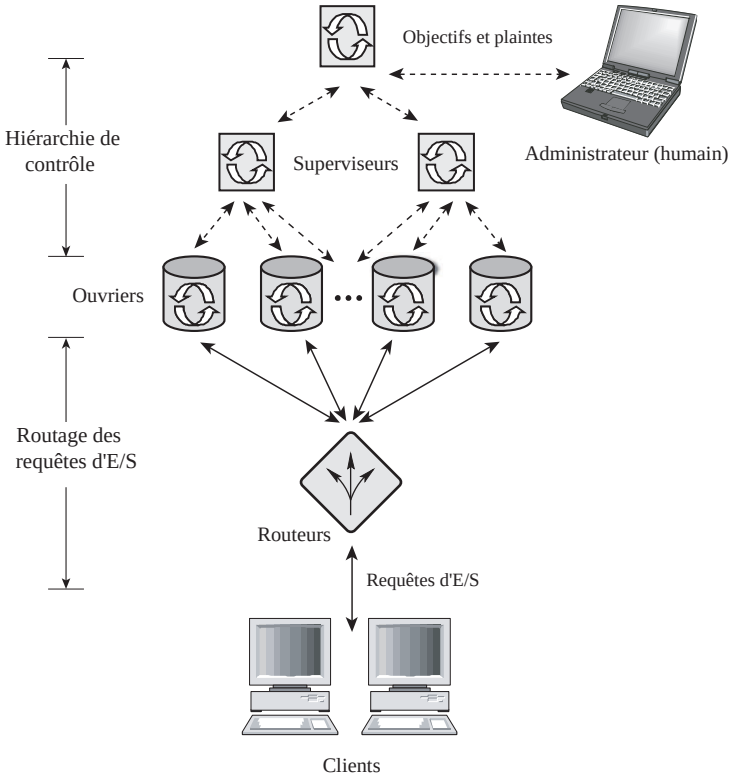


Figure 2.5 – Architecture de stockage autonome proposée par le projet Self-* (schéma extrait de [78])

2.7 Conclusion

Les besoins d'une application vis à vis du système de stockage sur lequel elle s'appuie sont caractérisés par plusieurs paramètres, dont notamment :

- la capacité de stockage (volume de données supporté),
- le taux de redondance des données, (immunité face à la perte de données),
- les performances d'accès aux données (débit et latence des opérations d'E/S),
- le maintien de la cohérence des données en cas d'accès concurrents,
- extensibilité et reconfiguration (possibilité de modifier les paramètres du système sans l'arrêter),
- la haute disponibilité des données (capacité du système de stockage à fournir ses services sans dégradation importante des performances malgré certaines pannes ou reconfigurations en cours),
- les mécanismes et garanties de sécurité offerts par le système.

Tous ces paramètres sont dépendants des choix opérés pour chacun des niveaux d'interface présentés au cours de ce chapitre et en particulier (1) des capacités physiques du matériel employé, (2) de la topologie physique et logique du système et (3) de la flexibilité et des optimisations offertes par les couches logicielles du système (GVL et SGF ou SGBD).

Reprenons maintenant en détails ces différents points dans le contexte des applications réparties s'exécutant sur grappes de stations.

- La capacité de stockage et le taux de redondance des données sont indépendants de la topologie du système et uniquement liés aux limites supportées par le matériel et les couches logicielles d'exploitation. Toutefois, le volume de stockage offert par une architecture répartie est potentiellement plus simple à étendre.
- Les performances d'accès aux données sont très fortement liées aux trois aspects suivants.
  - Les performances des réseaux d'interconnexion, des disques et plus généralement la configuration du (des) serveur(s) jouent bien sûr un rôle important.
  - La topologie du système a également un impact majeur, en particulier pour des charges d'E/S lourdes et la gestion d'un grand nombre de clients. Un serveur centralisé constitue un goulot d'étranglement car toutes les requêtes transitent par un nœud central et consomment ses ressources. Une architecture parallèle (par exemple un SGF parallèle ou un SGBD dupliqué/partitioné) résout partiellement ce problème car le serveur central n'est plus sollicité dans toutes les phases du traitement d'une requête et se voit limité à un rôle de gestion des métadonnées et/ou de routeur de requêtes. Une architecture à disque partagé va un cran plus loin en permettant d'utiliser plusieurs serveurs frontaux ; le goulot d'étranglement potentiel est alors déporté au niveau de l'infrastructure de partage (contrôleur SAN pour une implémentation matérielle, protocole maître-esclave pour une approche logicielle à la Petal). Enfin, une approche totalement décentralisée comme NASD ou FAB élimine tous les goulots d'étranglement au niveau de l'accès au données.
  - Les optimisations logicielles (notamment au niveau des politiques de cache) ont également un impact significatif.
- Les fonctionnalités de verrouillage et la sémantique de cohérence des données sont uniquement dépendantes du SGF[80]. A de rares exceptions près (NFSv4 et xFS, par exemple), la plupart des implémentations disponibles offrant des possibilités de verrouillage avancées et une sémantique de cohérence forte correspondent à des SGF à disques partagés.
- Les facultés d'extensibilité et de reconfiguration dynamique ne sont liées qu'aux capacités du matériel et des logiciels employés.
- Le critère de haute disponibilité est intrinsèquement lié à la topologie du système. Plus le système est décentralisé, plus sa tolérance aux situations exceptionnelles (pannes, lourdes actions de reconfiguration, pics de charge) est élevée. Les systèmes à disques partagés sont en ce sens les mieux armés, même dans le cas des architectures asymétriques, car chaque serveur peut potentiellement être remplacé par un autre.
- La sécurité d'un système de stockage est, quant à elle, inversement proportionnelle à sa nature décentralisée (en raison de la multiplication des points d'accès aux données). Dans le contexte des grappes, la sécurité n'est généralement pas une préoccupation majeure au niveau du système de stockage[81]. En effet, une grappe est généralement déployée dans un environnement relativement isolé de l'extérieur et sécurisé et les utilisateurs sont considérés comme dignes de confiance. Lorsqu'une grappe est partitionnée en plusieurs sous-grappes virtuelles, les principaux dispositifs de sécurité sont plutôt mis en œuvre au niveau des communications sur le(s) réseau(x) d'interconnexion. Par ailleurs, l'émergence des protocoles IP dans le monde des SAN

[80] Ce critère n'a pas autant d'importance dans le cas des SGBD car la quasi-totalité des systèmes garantissent une sémantique de cohérence forte et la concurrence est gérée différemment (via des mécanismes transactionnels).

[81] C'est la raison pour laquelle ces aspects n'ont pas été détaillés dans ce chapitre.

et l'arrivée des disques à objets permettront une plus grande flexibilité de configuration des politiques de sécurité au sein d'une architecture à disques partagés.

On peut globalement résumer l'adéquation entre les différents types de systèmes de stockage et les besoins applicatifs de la façon suivante :

- Un serveur centralisé a pour principal avantage d'être simple à administrer. Une telle architecture est bien adaptée à des charges modérées, sans concurrence d'accès, vers des données peu critiques. Par exemple, un serveur NFS s'avère souvent satisfaisant pour héberger les comptes des utilisateurs de la grappe d'un laboratoire. A l'opposé du spectre, certaines applications très exigeantes peuvent bénéficier d'une architecture monolithique hautement optimisée pour leurs contraintes (le parallélisme et la redondance de ressources étant dans ce cas gérés à l'intérieur du serveur). De telles machines sont cependant limitées à certaines « niches applicatives », en raison de leur coût d'acquisition et accessoirement de leur extensibilité limitée.
- Les architectures distribuées simples, basées sur des schémas (éventuellement combinés) de *striping*/duplication/partionnement permettent d'obtenir de bons niveaux de performances et de tolérance aux pannes. Ces solutions sont particulièrement adaptées à des charges d'E/S où les lectures sont majoritaires ainsi qu'à des problèmes faciles à paralléliser (tâches partitionnées sur des sous ensembles de données disjoints, sémantique de cohérence relâchée, etc.).
- Les architectures basées sur un partage d'accès, à bas niveau, à l'ensemble de l'espace de stockage sont complexes à mettre en ouvre et à optimiser. Elles disposent en contrepartie du plus fort potentiel de passage à l'échelle, de gestion de requêtes concurrentes, de reconfiguration dynamique et, en conséquence, de haute disponibilité. Ces critères sont cruciaux pour la majorité des applications de type « serveurs de données » et cette topologie a tendance à s'imposer comme épine dorsale des grands centres d'hébergement de données (*data centers*). À noter que, dans le cas d'un fonctionnement normal (pas de panne, ni de surcharge), une ferme de serveurs basée sur une infrastructure de stockage partagée a tout intérêt à être utilisée de façon partitionnée (via un routage adéquat au niveau de la couche d'admission des requêtes applicatives) pour éviter des phénomènes de contention sur les mécanismes de synchronisation. La nature partagée du système de stockage prend tout son avantage pour simplifier l'administration dynamique des serveurs : la réattribution des rôles des serveurs (en réaction à une panne ou à l'évolution de la charge d'entrée) est simple, transparente et efficace [197].

Les infrastructures de stockage réparties pour grappes atteignent actuellement une période charnière. Au niveau matériel, on observe une transition vers une utilisation généralisée de matériel banalisé (évolution des matrices de disques vers des grappes de briques de stockage, émergence d'Ethernet et des protocoles IP). L'intérêt pour les réseaux spécialisés (très faible latence, RDMA) et les disques rapides reste toutefois important pour certaines classes d'applications sensibles aux gains de performance fournis par ces technologies. D'autre part, l'échelle (volume de données, nombre et complexité des composants matériels et logiciels) des systèmes est en permanente expansion. En outre, les contraintes de haute disponibilité se resserrent (utilisateurs plus dépendants des données numérisées, plus exigeants...). En conséquence, l'administration de ces systèmes devient de plus en plus ardue, coûteuse et même pernicieuse (de nombreuses études montrent que la majorité des dysfonctionnements sont dus à des erreurs d'administration ; un cercle vicieux est donc en train de s'instaurer).

Le chapitre suivant décrit les travaux de recherche visant à répondre aux besoins importants en termes de flexibilité, de reconfiguration dynamique et d'auto-administration et situe le positionnement de notre proposition.

Chapitre 3

Positionnement de la contribution

Sommaire

3.1 Introduction

En décrivant la vaste gamme d'infrastructures de stockage réparti disponibles, le chapitre précédent a également mis en évidence la grande diversité des besoins applicatifs qui ont motivé leur développement. Toutefois, la plupart des systèmes existants souffrent des mêmes restrictions, à plusieurs niveaux, et ne répondent pas à l'ensemble des problèmes qui se posent aux administrateurs d'une grappe. Nous détaillons ci-dessous trois axes importants pour l'administration des systèmes de stockage : configuration flexible, reconfiguration dynamique et simplicité d'administration. Pour finir, un ensemble de contraintes communes à ces trois axes est brièvement présenté.

3.1.1 Configuration flexible

Les systèmes actuels, bien que construits selon un modèle en couches, présentent cependant un caractère relativement « monolithique ». Chaque couche implémente une fonction à très gros grain qui ne peut généralement pas être configurée finement sans induire d'importantes modifications du code (et n'exhibe souvent que peu de paramètres explicites permettant de configurer ses propriétés). Il est ainsi difficile d'optimiser un système de stockage pour un environnement donné (par exemple pour une technologie d'interconnexion précise) ainsi que pour un profil applicatif ciblé (si le système n'a pas expressément été développé pour ce type d'emploi). En outre, une telle structure ne facilite pas la réutilisation de code : à l'exception des services génériques de très bas niveau fournis par le système

d'exploitation, chaque implémentation d'une couche (telle qu'un SGF ou un GVL) doit réimplanter intégralement l'ensemble des services que l'on attend d'elle. De manière générale, ce manque de flexibilité constitue un frein à la maintenabilité et à l'évolutivité des systèmes de stockage.

3.1.2 Reconfiguration dynamique

De nombreuses applications industrielles ont des contraintes très fortes de haute disponibilité. Ces besoins s'accommodent mal d'éventuelles interruptions du service de stockage sous-jacent (et qui induisent parfois un arrêt complet des applications clientes). Or, pour fournir un fonctionnement satisfaisant, une infrastructure de stockage nécessite de fréquentes opérations d'administration. Pour atteindre des objectifs de haute disponibilité, la reconfiguration d'une couche du système de stockage se doit d'être *transparente* par rapport aux couches supérieures (applications incluses) :

- au niveau fonctionnel : les couches supérieures n'ont pas à être averties qu'une reconfiguration est en cours (et *a fortiori* à être arrêtées ou devoir prendre en compte une telle situation) ;
- au niveau des performances : l'ensemble des données stockées doit toujours être accessible et les performances d'accès à celles-ci ne doivent pas être excessivement dégradées.

La plupart des systèmes actuels offrent quelques possibilités de reconfiguration dynamique. Par exemple,

- un contrôleur RAID (logiciel ou matériel) gère la reconstitution des données suivant le remplacement d'un disque défaillant ;
- un GVL ou un DVR permettent généralement de modifier la structure de l'espace de stockage (ajout ou retrait de disques/volumes, changement du schéma de répartition des données, migration de données d'un périphérique à un autre) ;
- le nombre de clients d'un SGF réparti peut évoluer dynamiquement.

Il demeure cependant d'importants besoins qui ne sont pas ou seulement partiellement satisfaits par les infrastructures actuelles ; nous en décrivons ci-dessous les principaux.

Modification du code La possibilité de mettre à jour le code du système de stockage présente plusieurs avantages : suppression de bogues et de trous de sécurité, remplacement de l'implémentation d'une fonction par une version plus efficace, ajout de leviers pour rendre le comportement du système plus flexible, etc.

Extensibilité Dans le prolongement du point précédent, il est intéressant de pouvoir étendre les fonctionnalités du système après son déploiement : ajout de schéma de répartition des données, support de nouvelles stratégies et/ou d'interfaces réseau pour le transfert de données[1], etc.

Modification des paramètres du système Ce mécanisme permet d'influer sur des aspects tels que la politique d'ordonnancement des requêtes, les stratégies de transfert de données employées, la taille et la sémantique des caches afin d'optimiser le comportement du système selon l'évolution de sa charge et d'autres contraintes (nombre de clients, priorité respectives de ces derniers, etc.).

Modification de la structure du système Qu'il s'agisse de répondre à une panne ou de s'adapter à l'évolution des contraintes imposées par les clients, il peut être nécessaire d'effectuer de lourdes opérations de reconfiguration afin de modifier la structure du système de stockage. Par exemple, il peut être question d'ajouter des ressources physiques (disques, nœuds de stockage), de déplacer des données, de changer la façon dont celles-ci sont réparties, de déporter certaines tâches initialement affectées aux clients vers le(s) serveur(s) (ou vice versa).

[1] Ceci prend tout son intérêt avec les interfaces réseau récentes qui supportent une mise en service « à chaud » (*hot-plugging*), c'est-à-dire sans l'arrêt de la machine hôte) [182].

Instrumentation L'insertion temporaire de sondes peut faciliter la détection de bogues ou de goulots d'étranglement au sein du système. Cette approche est nettement préférable à une instrumentation statique car elle n'impose pas de surcoût dans le cadre d'un fonctionnement normal. De plus, des sondes peuvent être développées et ajoutées *a posteriori*, en fonction des besoins (il n'est pas nécessaire de prévoir toutes les fonctionnalités d'instrumentation lors du développement de la version initiale du système).

Les différentes opérations de reconfiguration qui viennent d'être énoncées sont complémentaires. Par exemple, la fonctionnalité d'instrumentation dynamique s'appuie sur la possibilité de modifier la structure du système (pour insérer des sondes) et/ou sur celle de mise à jour du code (pour remplacer une implémentation « normale » par une version instrumentée et vice versa). Inversement, l'instrumentation dynamique peut aider à détecter d'éventuelles lacunes du système, auxquelles on peut par la suite remédier par des opérations de mise à jour et/ou d'extension du code du système.

3.1.3 Simplicité d'administration

Les problèmes soulevés aux points précédents limitent la marge de manœuvre d'un administrateur. Le fait de lever ces contraintes ne peut cependant, à lui-seul, constituer un objectif suffisant. En effet, l'administration des systèmes actuels, dont la souplesse de paramétrage et de reconfiguration est encore généralement limitée, est dès à présent une source importante de difficultés financières et techniques. D'une part, même dans le cas d'une infrastructure de stockage haut de gamme, de nombreuses études industrielles estiment que les coûts humains d'administration d'une infrastructure de stockage sont quatre à huit fois supérieurs aux dépenses en matériels et logiciels associés. D'autre part, il devient également patent qu'une fraction majoritaire des dysfonctionnements sont liées à des actes humains erronés ; même des administrateurs expérimentés deviennent « dépassés » par la complexité des systèmes et l'exigence des contraintes qui leur sont associées. Pour ces raisons, le développement de systèmes flexibles et reconfigurables doit s'accompagner du souci de fournir des mécanismes automatisant les principales tâches d'un administrateur : paramétrage et optimisation de performances, supervision du système, prise de décisions et déclenchement d'une opération de reconfiguration.

3.1.4 Contraintes

Enfin, il convient d'insister sur certaines contraintes fortes qui accompagnent les attentes exposées ci-dessus.

Performances L'enrichissement du système en termes de fonctionnalités ne doit pas causer une dégradation importante de ses performances. Le système de stockage constitue souvent le goulot d'étranglement d'une infrastructure informatique et il n'est pas souhaitable d'accentuer cet état de fait.

Intégration avec l'existant La diffusion des innovations vers le monde industriel est relativement lente ; les changements d'interfaces et d'architectures sont très progressifs. L'adoption d'une interface novatrice telle que le stockage à objets (périphériques OSD) n'est pas encore une réalité, après presque dix ans de recherches. Aussi, il convient, dans la mesure du possible, de proposer des techniques qui peuvent s'intégrer aux produits industriels (systèmes d'exploitation et matériel) existants et bénéficier de leur évolution.

Grande diversité des besoins applicatifs Cette contrainte tend à privilégier une approche de type « boîte à outils » plutôt qu'une solution basée sur une topologie particulière.

La suite de ce chapitre est organisée comme suit. La section 3.2 présente un tour d'horizon des travaux autour de la construction de systèmes de stockage (re)configurables et autonomes. En sec-

tion 3.3, nous présentons notre contribution par rapport à l'état de l'art. Enfin, la section 3.4 décrit l'organisation des chapitres suivants, qui détaillent nos travaux.

3.2 Systèmes de stockage flexibles, reconfigurables et autonomes

Cette description de l'état de l'art est organisée selon trois axes :

- les canevas logiciels permettant de construire un système de stockage personnalisé par un assemblage de modules ;
- les systèmes adaptables, qui modifient leur comportement et/ou leur structure en fonction de leur charge ;
- les systèmes autonomes.

3.2.1 Systèmes de stockage configurables

De manière générale, la flexibilité de configuration d'un système est liée à son degré de modularité. Un exemple simple est celui du système d'exploitation K42 [13, 12], construit selon une architecture orientée objet. Au sein de K42, chaque ressource est représentée par une instance d'objet ; au niveau du stockage de données, chaque instance de fichier ouvert dispose de son propre cache de blocs et peut bénéficier d'une politique de gestion de cache (préchargement, éviction, synchronisation) adaptée à ses besoins. Outre le niveau de la représentation de ressources et des politiques de gestion associées, le principe de modularité peut s'appliquer à la manière de structurer le code du système de stockage. Deux exemples sont détaillés ci-dessous.

3.2.1.1 Systèmes de fichiers empilables

Depuis plus de quinze ans, de nombreux travaux de recherche ont étudié l'intérêt de développer des systèmes de gestion de fichiers de manière modulaire, avec pour principal objectif la réutilisation de code et la simplification du travail de développement. Le concept à la base des systèmes de fichiers empilables (*stackable file systems* ou encore *vnode stacking*) est simple : il s'agit d'étendre le principe de la couche d'abstraction VFS (cf 2.1.3) à une pile de couches fonctionnelles. Tous les modules d'une pile présentent la même interface (*celle du VFS*), identique vers le niveau inférieur et le niveau supérieur. Un exemple de pile simple est représenté en figure 3.1-a : une couche de compression et une autre de cryptographie ont été superposées et les fonctionnalités fournies par un SGF classique ont été éclatées en deux couches : un service de nommage hiérarchique (répertoires) et un service assurant le stockage sur un périphérique à blocs. La terminologie de « pile » est en fait trompeuse car il est possible de construire des configurations non linéaires : il peut y avoir plusieurs points d'entrée dans une pile (*fan-in*) et une couche donnée peut être empilée sur plusieurs sous-piles (*fan-out*). Une topologie *fan-in* est utile lorsque plusieurs types clients ont des besoins différents. Cette situation est illustrée en figure 3.1-b : des applications particulières (sauvegarde, archivage) n'ont pas besoin de manipuler les données sous forme décompressée et décryptée, un *fan-in* leur permet de « court-circuiter » les étages qui ne leur sont pas utiles. Un *fan-out* est principalement employé pour la duplication de données. Enfin, il est également possible de construire des SGF répartis (de type client-serveur) grâce à l'emploi d'une couche générique de transport, dont le seul rôle est le support des appels RPC (cette couche est en fait déployée à la fois au niveau du client et à celui du serveur). La sémantique du SGF réparti est uniquement implémentée au niveau des couches coopératives « client » et « serveur ». La figure 3.1-c illustre la construction d'un SGF réparti, combiné au principe de *fan-out* pour dupliquer les données localement.

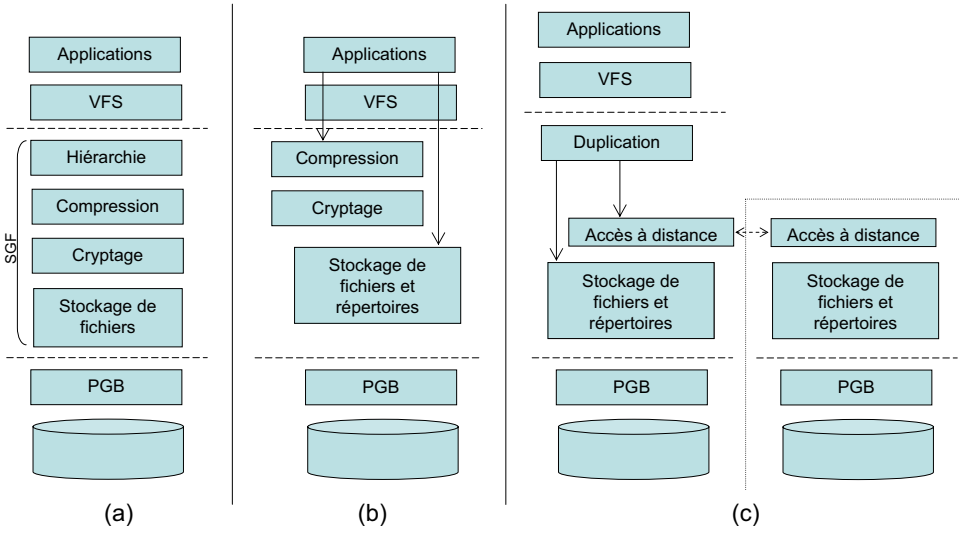


Figure 3.1 – Exemples de systèmes de gestion de fichiers construits par empilement de modules

Les travaux les plus poussés sur les SGF empilables ont été menés à UCLA (*University of California - Los Angeles*) au début des années 90 [93, 94], et ont notamment été à la source des contributions suivantes.

- L'introduction d'une interface extensible permet l'ajout de nouvelles opérations. La seule règle de programmation imposée par l'extensibilité du modèle est que chaque module doit fournir un traitant pour les opérations qu'il ne peut prendre en compte. Généralement, ce traitant consiste simplement à faire suivre la requête inchangée vers la couche de niveau inférieur.
- L'absence de contraintes par rapport à l'espace d'adressage associé à chacun des modules. Chaque module peut éventuellement être exécuté dans un domaine de protection différent sans aucune modification de code. Une telle configuration induit un surcoût non négligeable mais offre plus de souplesse. Ainsi, une portion de pile (voire l'intégralité) peut être exécutée en mode utilisateur, ce qui permet de simplifier le débogage (avant une intégration dans le noyau) et s'adapte bien avec une architecture de type micro-noyau.
- Le développement d'une infrastructure assurant la cohérence des informations (données et méta-données) cachées par différents modules au sein d'une pile. Cette infrastructure repose sur un gestionnaire de cache centralisé connu de tous les modules et prend en compte le cas complexe dans lequel un module (tel qu'une couche de compression) modifie la sémantique des informations manipulées.
- Une évaluation précise, à la fois en termes de génie logiciel (simplicité d'assemblage d'une pile et d'ajout de nouveaux modules, même par des programmeurs peu expérimentés) et de performances (surcoût de l'ordre de 3 à 10% par rapport à une implémentation monolithique) a permis de valider l'approche avec de nombreuses configurations de piles.

Plus récemment, des contributions de l'Université Columbia ont porté sur l'utilisation de patrons de conception [236] et de langages spécialisés [237] pour simplifier significativement le développement des modules d'un SGF empilable, améliorer leur portabilité et éviter toute modification du système d'exploitation hôte. Les évaluations ont montré des gains importants en temps de développement et de portage, ainsi qu'un surcoût réduit (0,8 à 2%).

3.2.1.2 Swarm

Swarm [91, 155] est un projet de l'Université d'Arizona (mené entre 1998 et 2000) par des chercheurs impliqués précédemment dans d'autres travaux du cru : Zebra (cf 2.4.3.3), Scout et x-kernel (voir 4.6), dont de nombreuses idées ont été reprises.

Swarm est un canevas permettant de construire des systèmes de stockage de bas niveau (potentiellement répartis). Il s'agit d'une infrastructure configurable et extensible, qui n'impose pas de protocole ni d'abstraction particuliers pour l'accès aux données par les clients. Un système construit avec Swarm est basé sur le principe d'un empilement de modules. Swarm n'est à proprement parler ni un SGF réparti, ni un disque virtuel réparti mais permet d'implémenter ce type de systèmes. Ceci témoigne de la volonté des auteurs de ne pas imposer systématiquement la complexité et le surcoût liés à la synchronisation de telles architectures réparties. Swarm peut également être employé pour déployer un système de stockage réparti sans partage de données entre clients ou encore un système de stockage centralisé.

Contrairement aux *systèmes de fichiers empilables* présentés en 3.2.1.1 qui imposent une interface unique à tous les niveaux (celle présentée aux clients — à l'extrémité supérieure — et l'interface entre chacun des modules du système) un système de stockage construit avec Swarm n'impose pas d'interface particulière à un niveau donné. En d'autres termes, l'interface exportée vers une couche « cliente » est libre (fichier, bloc, ou potentiellement n'importe quoi d'autre) et une pile de modules Swarm peut effectuer une ou plusieurs *traductions d'interface*.

Même si aucune interface n'est imposée aux couches clientes, le coeur de Swarm repose sur la notion de *striped log*, inspirée du SGF réparti Zebra : répartition sur plusieurs disques (selon un schéma RAID) du contenu d'un tampon (*log*) géré à la manière d'un LFS (la seule opération étant *append*, c'est-à-dire l'ajout de données en fin de tampon, cf 2.1.2.3). Comme pour Zebra et xFS, cette technique permet d'agréger les petites requêtes d'écriture (et donc d'augmenter leur performance), de décharger les serveurs de la majorité des opérations (chaque nœud client à son propre tampon et effectue les calculs de parité et les découpages de données nécessaires. En outre, dans un schéma d'utilisation simple (sans partage de données entre clients), aucune coordination entre serveurs n'est nécessaire. Il en va de même pour les clients dans le cas d'une reconstruction de données après la panne d'un serveur.

Pour simplifier et enrichir la vue offerte aux couches clientes, d'autres modules (nommés *services*) ont été développés au dessus des modules Swarm qui implémentent la notion de *striped log*. Chaque service permet d'étendre et/ou de masquer et d'abstraire les fonctionnalités du service directement sous-jacent. Par exemple, le service "disque logique" (*logical disk*, inspiré de [61], fournit une interface classique exportant un ensemble fini de blocs à accès aléatoire qui masque l'interface moins habituelle de *log* infini, modifié par opérations d'*append*).

L'extrémité inférieure d'un système Swarm repose sur le module « disque » (*disk layer*), qui reçoit des requêtes en termes de fragments de log (lecture, écriture, suppression) et les convertit en requêtes disque (lecture et écriture de blocs), grâce à une table de correspondance qu'il maintient. La répartition est gérée de manière transparente au niveau d'un nœud client, grâce à un module de transport offrant la même interface que le module disque qui propage les requêtes vers le nœud serveur hébergeant le module disque concerné[2].

Les deux principaux systèmes construits à partir du canevas Swarm furent deux systèmes de fichiers centralisés (au sens où deux nœuds distincts ne peuvent pas partager de données) mais dont les données peuvent éventuellement être stockées sur un ensemble (éventuellement commun) de nœuds serveurs, afin d'augmenter le parallélisme d'E/S et la tolérance aux pannes. Le premier système,

[2]Le protocole réseau et la ou les stratégies de transfert utilisées pour les requêtes n'ont pas été documentés.

Sting, est un LFS « classique » ; le second, ext2fs/Swarm, est une adaptation transparente du SGF local ext2, pour bénéficier des avantages d'un LFS. L'architecture de ces deux systèmes est représentée en figure 3.2. Les deux cas présentent un ensemble de modules en commun :

- au niveau d'un serveur (dans le cas d'un déploiement réparti), le module disque, précédé d'un éventuel module de sécurité (protection des fragments exportés par le serveur via une liste de contrôle d'accès) ;
- au niveau d'un client, les briques de base qui implémentent l'abstraction de *striped log* :
  - le module *Striper*, implémente le striping des fragments de log vers les serveurs de stockage ;
  - le module *Parity* gère le calcul des informations de parité associées à un *log* et l'éventuelle reconstruction des données en cas de panne de l'un des disques ;
  - le module *Log*, implémente l'abstraction de *log*, une suite infinie de blocs et d'enregistrements immuables.
  - le module *Cleaner* effectue une récupération périodique de l'espace inutilisé au sein d'un log afin de masquer sa taille bornée (par la capacité des disques sous-jacents) auprès des services de niveaux supérieurs.

Le service *StingFS* est un SGF local qui traduit les requêtes de niveau fichier émanant de la couche VFS en requêtes de niveau log. Dans le cas de ext2fs/Swarm, le code du SGF ext2 n'est pas modifié. Le service *SwarmLD* exporte un périphérique bloc (PGB) vers le système d'exploitation. Le service disque logique sert de point de conversion entre l'interface bloc du service supérieur et l'interface log de la couche inférieure.

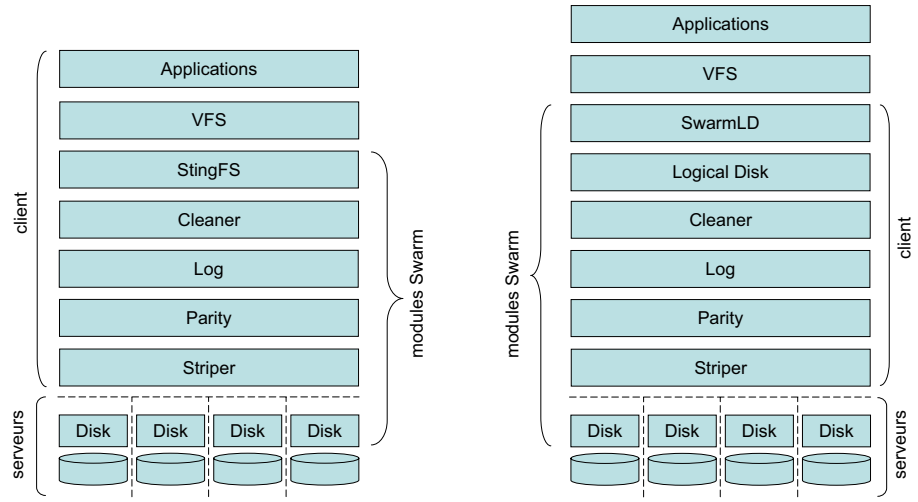


Figure 3.2 – Exemples de systèmes de stockage construits avec Swarm : Sting et ext2fs/Swarm

Le bilan de Swarm est mitigé. D'un côté, ces travaux ont fait avancer l'intérêt des systèmes de stockage modulaires en développant un modèle plus flexible et général que ceux définis précédemment (tels que les SGF empilables), notamment pour la liberté offerte pour le choix des interfaces et des mécanismes de synchronisation. Cependant, Swarm n'a pas concrétisé toutes ses ambitions. La

majeure partie des travaux s'est concentrée sur des aspects d'optimisation et de réingénierie autour des LFS et du concept de *disque logique*. Il y a peu de mesures de performances et aucune évaluation du surcoût causé par la modularité (par rapport à un LFS « monolithique »). L'implémentation et les expériences dans un contexte réparti furent assez limitées et n'ont pas pleinement exploité la liberté offerte par le modèle initial. En particulier, elles n'ont visé que des topologies sans partage (support de plusieurs clients par les nœuds serveurs mais sans possibilité de partager les données entre clients), qui ne présentent que peu d'intérêt par rapport aux principaux besoins applicatifs. Enfin, les volontés initiales de développer des méthodes de cache (le long d'une hiérarchie de modules ainsi qu'au niveau d'un cache coopératif), le support optimisé d'applications parallèles et la gestion (éventuellement simultanée) de multiples protocoles d'accès aux données (par exemple NFS, un SGF parallèle ou encore HTTP)[3], n'ont, à notre connaissance, jamais connu de suite concrète.

Au strict niveau du modèle architectural développé par le projet Swarm, on peut, en outre, formuler les remarques et critiques suivantes :

- Par manque de documentation sur le sujet, il est difficile de déterminer quelles sont les contraintes sur la communication entre deux modules[4] et, par extension, quel est le modèle de programmation d'un module Swarm.
- La modularité et la configurabilité des fonctions placées sur le serveur sont restreintes. L'architecture vise délibérément à placer la quasi-totalité des fonctions du système de stockage au niveau du client. Les modules déployés au niveau des serveurs sont limités à une interface de bas niveau (fragments de log).
- Plus généralement, le modèle de Swarm a pour pierre centrale l'abstraction de *striped log*, ce qui restreint considérablement la gamme de systèmes pouvant être construits. Ce choix apparaît relativement radical, car même si l'intérêt des optimisations apportées par les LFS a été démontré, celles-ci ne sont pas vitales pour construire un système de stockage efficace et l'abstraction du *log* est relativement déconcertante et difficile à maîtriser pour d'éventuels programmeurs désireux d'enrichir la bibliothèque de modules du canevas.
- Enfin, aucun aspect ayant trait à la reconfiguration dynamique n'a été considéré.

3.2.2 Systèmes adaptables

Les travaux décrits dans cette section ont pour objectif principal la construction de systèmes capables de s'adapter à différentes contraintes en cours d'exécution. Le premier exemple est une infrastructure permettant d'optimiser les performances d'applications intensives en E/S en fonction de l'évolution de la charge qu'elles génèrent. La seconde partie présente des outils de maintenance conçus pour effectuer des reconfigurations dynamiques sans impact sur la qualité de service perçue par les clients.

3.2.2.1 Abacus

Abacus [8, 6], développé à Carnegie Mellon University, est une infrastructure logicielle pour le support d'applications en grappes de type « fouille de données » (*data mining*). Initialement développé au niveau d'un SGF réparti déstiné à l'architecture NASD (cf 2.4.5), Abacus a ensuite été

[3] Hormis dans le cas de données en lecture seule exportées vers un ensemble de clients très hétérogènes, il nous est difficile d'imaginer une motivation pertinente pour l'exportation simultanée de données via de multiples interfaces, d'autant plus que le maintien de la cohérence des données risque de devenir très complexe dans un tel cas.

[4] et par incidence sur les protocoles de communication entre client et serveur.

étendu pour le support des applications elles-mêmes. Son objectif principal est l'optimisation automatique du placement des tâches nécessaires à l'extraction des données ; celui-ci est motivé par plusieurs constats.

- Un choix de placement bien adapté à une application dépend de nombreux paramètres et se révèle souvent difficile et long à déterminer, même pour un administrateur expérimenté.
- Si certains paramètres sont relativement statiques, tels que la configuration matérielle de la grappe (puissances relatives des différents nœuds, topologie et capacité du réseau d'interconnexion, ...), d'autres ne peuvent être déterminés qu'au moment de l'exécution (utilisation concurrente des ressources de la grappe par d'autres applications, voire concurrence d'accès au même ensemble de données).
- Même en l'absence de concurrence, une application peut voir certains de ses paramètres varier considérablement à l'exécution : une longue hiérarchie de filtres induit une consommation de ressources très variable en fonction du jeu de données qu'elle traite, une même application peut être structurée selon plusieurs phases d'exécution aux profils très variés, etc.

L'approche d'Abacus consiste donc à confier le travail d'optimisation au système plutôt qu'au programmeur ou à l'administrateur et à effectuer ce travail de manière dynamique, en modifiant périodiquement le comportement du système en fonction de ses contraintes. Cette approche a aussi le mérite d'être plus générique : elle peut être appliquée sans effort à différentes applications et permet également d'améliorer l'utilisation globale des ressources d'une grappe où cohabitent plusieurs applications.

De manière plus précise, Abacus est à la fois un modèle de programmation et un environnement d'exécution (*run-time system*) et joue deux rôles : déterminer automatiquement (et dynamiquement) la meilleure stratégie de placement de tâches et effectuer les migrations de tâches nécessaires.

Modèle de programmation Un système de stockage (ou une application) est modélisé(e) comme un graphe de tâches réparties entre un client et un ou plusieurs serveurs. Chaque tâche est implémentée par un composant et correspond à une fonction d'une granularité moyenne (calcul de parité, cache, filtre, agrégation, ...). La plupart des composants sont mobiles et définis comme tels : ils peuvent être déplacés d'un nœud à un autre selon les décisions du système de supervision. En conséquence, chaque composant doit fournir deux méthodes (sérialisation et désérialisation) permettant de sauvegarder et de restaurer son état. Les seuls composants non migrables sont ceux employés aux extrémités de la chaîne de traitement : un composant *console* sert de point d'ancrage avec le système d'exploitation du client et un composant *stockage* est lié au support local de stockage d'un serveur. De plus, un programmeur ou administrateur peut éventuellement choisir de restreindre la mobilité d'autres types de composants.

Dans le cas général, les composants interagissent via une chaîne d'appels synchrones démarrée au niveau du composant console par l'arrivée d'une nouvelle requête. Cependant, un composant peut aussi devenir actif (et être ainsi à l'origine d'une chaîne d'appels) spontanément dans certains cas : par exemple, un composant implémentant un cache *write-back* synchronise périodiquement les données sur disque.

Enfin, le modèle de programmation requiert que l'application soit structurée selon un modèle itératif, où les données sont amenées au client de manière progressive ; si l'ensemble des données sont réclamées et transférées d'un coup, l'environnement de supervision ne pourra pas collecter les statistiques fines nécessaires à ses calculs d'optimisations. Cette contrainte est naturellement satisfaite par la plupart des applications visées (ne serait-ce qu'en raison de la taille de l'ensemble des données).

Environnement d'exécution Deux entités d'administration sont déployées sur chacun des nœuds : (a) un gestionnaire de liaisons, en charge de l'instantiation des composants, de l'invocation (éventuellement distante, de façon transparente) de leur méthodes et de leur migration et (b) un gestionnaire de supervision qui collecte des statistiques sur l'utilisation des ressources et s'en sert pour décider d'éventuels déplacements de composants.

Les communications entre deux composants s'exécutant dans le même espace d'adressage sont réalisées par un simple appel de procédure et les données sont transmises sans recopie. Deux composants distants communiquent par appels de procédures à distance. L'environnement d'exécution prélève ses statistiques à l'occasion de chaque appel de méthode d'un composant.

Pour chaque instance de composant, le gestionnaire de ressources maintient la liste des ressources système consommées, y compris le temps CPU par les différents threads l'ayant traversée. Plus précisément, le gestionnaire de ressources utilise quatre principaux indicateurs pour effectuer ses prédictions : quantité de données produites/transférées par un composant, consommation mémoire, nombre d'instructions exécutées par octet transféré, temps d'attente dans un objet.

Ces informations sont prélevées et affinées sur chaque nœud au cours d'un intervalle de temps fixe (de l'ordre d'une seconde), collectées par chaque serveur, et utilisées comme paramètres d'entrée d'un modèle analytique qui évalue le gain potentiel des autres stratégies de placement disponibles (en prenant en compte le surcoût causé par les différentes étapes d'une migration). La décision de déplacer un ou plusieurs composants est prise uniquement si le bénéfice net attendu dépasse un certain seuil (généralement 30%) afin d'éviter de mauvais choix dus à d'éventuelles imprécisions de mesure et/ou du modèle. Afin de maintenir un fonctionnement simple passant à l'échelle, les gestionnaires de ressources de différents serveurs ne communiquent pas entre eux pour déterminer une stratégie globale à la grappe ; pour ses estimations, un serveur ne considère que les flux de communication qui lui sont connectés.

Pour effectuer une migration, le gestionnaire de liaisons commence par bloquer toute nouvelle tentative d'appel de méthode sur le composant concerné et attend que toutes les exécutions en cours se terminent. L'état du composant est alors sérialisé, transmis au nouveau nœud qui doit l'héberger et restauré. Les tables de localisation de composants maintenues par le gestionnaire de liaisons sont mises à jour sur les nœuds source et destination, et l'invocation de méthodes est finalement réactivée.

La figure 3.3 présente l'architecture de l'environnement Abacus ainsi qu'un exemple de configuration simple (client et serveur uniques, pas de partage de données ni de concurrence d'accès). A l'exception des extrémités d'un assemblage (représentées en gris foncé), les composants peuvent être déplacés d'un nœud à l'autre, selon les décisions du système de supervision. Une requête issue d'une application cliente engendre un chaîne d'invocations synchrones qui traverse la chaîne d'instances de composants (mais pas forcément de bout en bout — cette chaîne peut par exemple s'arrêter au niveau du cache). L'infrastructure de supervision intercepte tous les appels de méthodes entre composants.

Un prototype d'Abacus a été développé en C++ et évalué dans le cadre de nombreux exemples réalistes. Les expériences ont montré qu'un tel système est capable de corriger des choix initiaux peu efficaces, de déterminer la meilleure configuration possible en fonction des contraintes présentes à l'exécution et d'améliorer significativement les performances du système (d'un facteur deux à dix par rapport à un placement statique). Nous décrivons ci-dessous plusieurs situations concrètes pour lesquelles Abacus a démontré tout son intérêt :

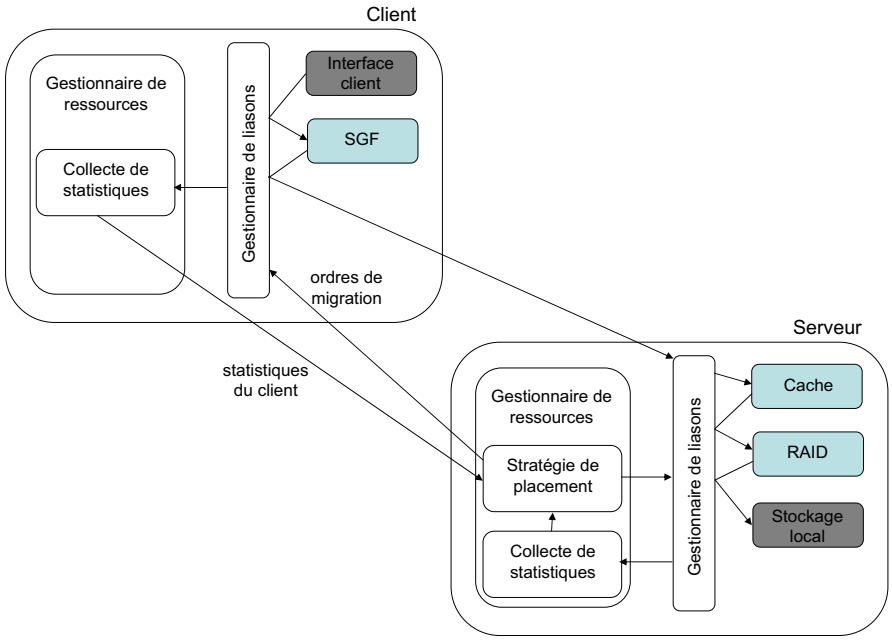


Figure 3.3 – Exemple de système de stockage déployé au sein de l'environnement Abacus

- adaptation aux caractéristiques statiques (topologie, capacité) et dynamiques (congestion) du réseau d'interconnexion ;
- adaptation au profil des requêtes d'une application et à la contention sur des données partagées : en fonction du taux de « réutilisation » des données par les clients et du taux d'accès de modification concurrentes envers des informations partagées, le système détermine s'il est plus avantageux de maintenir un cache au niveau des clients plutôt qu'à celui du serveur ;
- adaptation à différentes phases applicatives ;
- adaptation pour un partage optimal des ressources d'un serveur entre plusieurs clients : le serveur privilégie l'accueil des tâches qui en tirent le meilleur bénéfice.

En contrepartie, l'infrastructure d'Abacus impose un surcoût temporel non négligeable dû principalement à l'utilisation du CPU pour la supervision et le calcul des nouvelles stratégies de placement (la consommation en mémoire est relativement faible). Le ralentissement imposé par l'infrastructure est généralement de 10%, lorsque l'application s'exécute pendant un long intervalle de temps et/ou s'accommode bien des adaptations dynamiques effectuées par Abacus ; dans le cas contraire le surcoût peut atteindre 25%.

Si les contributions d'Abacus sont indéniables et significatives, on peut néanmoins remarquer certaines limites.

- Le modèle architectural présente plusieurs restrictions. Il est, tout d'abord, centré autour d'une topologie de type « clients-serveurs » et ne prend pas en compte des modèles plus complexes (par exemple, une architecture à plusieurs étages avec un serveur central servant d'intermédiaire entre les clients et un ensemble de nœuds de stockage). Le transport sur le réseau des requêtes et des données n'est pas modélisé de façon explicite ; rien n'est prévu pour optimiser le protocole de communication à l'intérieur d'Abacus. Plus généralement, le seul levier d'adaptation

disponible est le placement de tâches sur la grappe ; la possibilité d'ajuster dynamiquement le comportement des différents composants n'a pas été envisagée.

- L'infrastructure est adaptable, mais pas véritablement reconfigurable. Il n'est pas possible de modifier l'implémentation d'un composant ni de déployer de nouveaux types de composants de manière dynamique. Il n'est pas non plus possible de désactiver le système de supervision (dont le coût n'est pas négligeable). Cette fonctionnalité pourrait pourtant être avantageuse dans certains cas : si l'on atteint un « régime stationnaire » avec un placement optimal des tâches, peu de concurrence entre clients et applications et/ou peu de variations de charge, alors la collecte de statistiques avec une granularité et un taux d'échantillonnage fins perd de son intérêt. Il est, en outre, impossible de déplacer la fonction de serveur d'un nœud à un autre de façon transparente.

  Par extension, les modifications structurelles sont restreintes au déplacement de composants, alors que d'autres opérations pourraient fournir des bénéfices complémentaires. En cas de concurrence entre plusieurs clients, il s'avère souvent avantageux de déplacer au niveau du serveur les composants associés à des fonctions de cache ou de gestion de répertoire pour limiter la contention sur les informations partagées. Cette approche permet de limiter le trafic réseau causé par des messages d'invalidation de cache ou des tentatives réitérées d'acquisition de verrous mais elle ne permet qu'un gain limité car il y a toujours un phénomène de contention au niveau du serveur, dû à la concurrence entre les multiples instances (une par client) d'un même type de composant. Une solution plus efficace consisterait à fusionner ces différentes instances lorsqu'elles se retrouvent groupées au niveau du serveur.

3.2.2.2 Outils de reconfiguration non intrusifs

Récemment, plusieurs travaux ont cherché à produire des outils d'administration plus adaptables ; nous en présentons ici deux brefs exemples.

Acqueduct [137] est un gestionnaire de volumes logiques pour SAN capable d'effectuer des opérations de reconfiguration dynamique (modification de la structure physique des volumes, déplacement de données, etc.) de manière efficace, tout en fournissant des garanties (statistiques) de qualité de service aux applications clientes. Il est basé sur une boucle de contrôle permettant d'ajuster la cadence d'une opération de reconfiguration en fonction de l'évolution des latences perçues par les clients.

Une infrastructure pour simplifier le développement d'outils de maintenance à également été proposée [223]. De nombreuses tâches d'un système d'exploitation induisent des transferts de données fréquents et/ou volumineux mais avec peu de contraintes temporelles et d'ordre (sauvegarde, recherche de virus, indexation de fichiers, défragmentation d'un SGF, etc). L'infrastructure développée par CMU est intégrée au noyau du système d'exploitation et fournit une interface explicite qui permet aux services système et aux applications de déclarer les requêtes d'E/S associées à des tâches de fond comme telles. Le système d'exploitation (modifié au niveau du pilote de disque) peut alors profiter du relâchement des contraintes sur les E/S associées à une reconfiguration pour effectuer un ordonnancement optimal. Cette stratégie permet d'augmenter les performances des tâches de fond, sans causer d'impact important sur l'exécution des applications (ralentissement inférieur à 2%).

3.2.3 Systèmes autonomes

Les recherches sur les systèmes de stockage autonomes ne sont encore qu'embryonnaires et se sont jusqu'à présent concentrées sur des aspects de haut niveau. La plupart des travaux ont développé

des méthodes d'inférence et d'apprentissage permettant à un système d'améliorer la pertinence de ses observations et la validité de ses prédictions. En voici deux exemples :

- la modélisation automatique d'un disque dur et la détermination automatique du placement des données sur le disque en fonction du profil des requêtes d'E/S [98] ;
- l'inférence automatisée des propriétés d'un fichier (accès en lecture seule, durée de vie très courte, ...) en fonction de son nom et de ses attributs afin de lui associer une politique de gestion adaptée (duplication pour fichiers souvent ouverts en lecture seule, placement en NVRAM pour petits fichiers à très courte durée de vie, etc) [140].

On peut voir aussi Abacus comme un embryon de système autonome car il simplifie la tâche d'un administrateur en automatisant des décisions de reconfiguration et leur mise en œuvre mais le système de supervision n'est pas reconfigurable ni extensible, et le spectre des opérations de reconfiguration possibles est limité.

3.3 Bilan et proposition

Ce chapitre a présenté les limites des systèmes de stockage courants en termes de flexibilité, reconfiguration dynamique et gestion autonome puis exposé l'état des recherches actuelles dans ces domaines.

Il y a toujours beaucoup d'intérêt (industriel et académique) autour des infrastructures de stockage réparti de bas niveau (bloc ou objet) comme l'a montré le chapitre précédent. Les efforts ont été focalisés sur le développement de nouvelles architectures mais peu de travaux ont cherché à fournir des mécanismes génériques pour simplifier le développement et l'administration de tels systèmes. Cet état de fait est rédhibitoire pour plusieurs raisons :

- la grande diversité des besoins ;
- la complexité de mise au point et d'optimisation d'un logiciel de stockage de données ;
- le caractère critique d'une infrastructure de stockage sur le fonctionnement, les performances et la disponibilité d'un système informatique.

Des projets comme Swarm et Abacus ont apporté des contributions significatives dans la voie qui nous intéresse mais aucun d'eux ne satisfait tous les objectifs au niveau de la flexibilité et de la reconfiguration dynamique. En outre, les aspects liés au comportement autonome n'ont, à notre connaissance, été que peu étudiés dans le contexte des couches basses d'un système de stockage.

Nos travaux ont pour ambition de contribuer à remédier aux lacunes mentionnées ci-dessus. Nos objectifs précis sont multiples.

- Faire évoluer les idées développées dans le cadre de Swarm et Abacus pour aboutir à une boîte à outils permettant de créer, par assemblage de composants, les briques de base nécessaires à une grande variété d'infrastructures de stockage réparti (disque virtuel réparti, SGF partagé, serveur central relié à une ferme de nœuds de stockage).
- Offrir une modélisation explicite de la structure répartie d'un système ainsi créé afin de permettre diverses adaptations spécifiques à la technologie d'interconnexion qui équipe la plateforme de déploiement. En contrepartie, les composants n'étant pas impliqués explicitement dans les mécanismes de communication réseau doivent pouvoir être développés de façon indépendante du matériel ciblé.

- Fournir un ensemble de mécanismes génériques de reconfiguration dynamique satisfaisant les attentes détaillées en section 3.1. Développer également les fondations requises pour l'introduction progressive d'aptitudes d'auto-administration dans le système. Pour atteindre ces objectifs, il paraît nécessaire de développer un système réflexif, c'est-à-dire capable de s'introspecter, d'appréhender et de modifier sa structure et son fonctionnement.
- Comme mentionné plus haut, notre proposition a pour vocation de simplifier la construction d'une large gamme de systèmes de stockage. Nous nous intéressons à la fois aux architectures matérielles dédiées au rôle du stockage de données (cas d'une matrice de disques décentralisée à la FAB, cf. 2.6.1) ainsi qu'à des grappes de machines utilisées comme un système à image unique. Dans ce dernier contexte, nous cherchons à évaluer la viabilité d'une architecture en grappe où les ressources locales de stockage sont mutualisées, chaque nœud jouant à la fois un rôle au niveau applicatif et offrant également une partie de ses ressource d'exécution pour répondre aux requêtes d'E/S de ses pairs. Au passage, nous étudions l'adéquation de cette approche avec plusieurs types d'optimisations matérielles : transferts directs en mémoire distante (RDMA) et processeurs multithreadés (SMT).

3.4 Organisation de la contribution

La suite de ce document est structurée de la façon suivante :

- le chapitre 4 décrit les principes du nouveau canevas logiciel que nous avons introduit pour simplifier la programmation d'infrastructures de stockage configurables et extensibles ;
- le chapitre 5 expose le modèle employé pour abstraire les stratégies de transfert de données développées pour différentes plates-formes ;
- le chapitre 6 détaille un ensemble de mécanismes génériques de reconfiguration dynamique et une architecture de contrôle posant les bases nécessaires à une gestion semi-automatisée d'une infrastructure de stockage ;
- le chapitre 7 est centré sur l'évaluation des performances de notre prototype ;
- la conclusion dresse un bilan de nos travaux et traite des perspectives qu'ils ouvrent.

Chapitre 4

Un canevas logiciel flexible pour la construction de systèmes de stockage répartis

Sommaire

4.1 Introduction

Ce chapitre expose notre proposition de canevas, nommée *Proboscis*[1], visant à combler les lacunes identifiées précédemment. Il détaille notamment les modèles de composition et d'exécution employés ainsi que le processus de déploiement d'un système de stockage. Les protocoles optimisés pour le transfert de données sont décrits au chapitre 5. Les détails spécifiques aux mécanismes de reconfiguration dynamique et à l'architecture d'administration font l'objet du chapitre 6. Le présent chapitre évoque toutefois quelques considérations liées aux transferts de données et aux besoins en matière de reconfiguration afin de justifier certains choix de conception.

Proboscis regroupe un modèle de composants spécialisé pour la construction de systèmes de stockage répartis, une bibliothèque de composants prédéfinis et une infrastructure d'exécution qui fournit un ensemble de services (dont l'allocation de ressources d'exécution) aux composants déployés.

Le modèle de composants et le modèle d'exécution de Proboscis sont respectivement introduits en 4.2 et en 4.3. La section 4.4 décrit le processus de déploiement d'une configuration. Les mécanismes avancés de routage réseau réalisables avec Proboscis sont présentés en 4.5. Enfin, nous discutons les différentes caractéristiques de notre proposition en 4.6 et dressons le bilan de ce chapitre en 4.7.

4.2 Modèle de composition

Le modèle de composition sur lequel repose Proboscis se veut simple mais puissant en termes d'expressivité. Cette partie présente d'abord la notion centrale de « chemin », puis la granularité et les interfaces d'un composant. Elle décrit ensuite les différents types de composants, les règles de composition à respecter et les aspects relatifs à la liaison de deux composants.

Au sein de ce chapitre et des suivants, nous utilisons les termes *composant* et *extension* de manière interchangeable pour désigner l'élément de base du modèle de structuration.

4.2.1 Concept de « chemin »

Notre canevas est spécialisé pour la construction de services de stockage de bas niveau, dont le rôle principal est le routage de requêtes d'accès à un ensemble d'organes de stockage. Pour cette raison, notre modèle de composition est relativement simple : un service de stockage peut être décrit comme une chaîne de composants de traitement, reliant un client (qui lit et/ou écrit des données) à un ensemble de serveurs qui assurent la fonction de stockage. Chaque service déployé correspond à un assemblage d'instances de composants qui prend la forme d'un pipeline arborescent[2]. Il n'y a pas véritablement d'encapsulation globale du service au sein d'un composant composite. Cette structure est délibérée : notre objectif principal est de représenter la chaîne de traitement par laquelle transitent les requêtes d'accès aux données, ce que nous appellerons désormais un *chemin*.

La notion de chemin est récursive : un chemin est constitué d'un assemblage de sous chemins, eux-mêmes décomposables en sous-parties jusqu'au niveau élémentaire des composants. Nous utilisons par la suite le mot *chemin* de façon générique, à la fois pour désigner un chemin *complet*

[1] Il s'agit d'un terme biologique employé pour décrire une longue structure souple, permettant d'aspirer ou de projeter diverses substances. Il correspond en particulier à la trompe d'un éléphant ou d'un insecte (moustique, papillon), et peut aussi désigner l'appendice nasal volumineux d'un humain. Dans notre contexte d'étude, ce nom illustre l'idée d'une longue trompe, souple et réactive, permettant des échanges de données optimisés entre plusieurs nœuds d'une grappe.

[2] Dans certains cas rares, la structure construite correspond à un treillis plutôt qu'à une arbre. Voir par exemple la configuration décrite en 4.5 et représentée en figure 4.7.

(l'intégralité de la chaîne de traitement qui relie un client à un serveur) ou une sous-partie de celui-ci. Le contexte narratif suffit généralement pour marquer une distinction, lorsque cela est nécessaire. Notre concept de chemin sera étayé au fil de ce chapitre. Pour terminer cette entrée en matière, nous décrivons quelques caractéristiques complémentaires des *chemins* au cœur de notre canevas.

Parcours bidirectionnel Une requête parcourt un chemin dans un sens (émission) puis dans l'autre (acquittement) Le fait d'utiliser le même chemin dans les deux directions simplifie la mise en œuvre des chemins et limite le nombre d'instances déployées (et la consommation de ressources qui en découle). Cela permet en outre aux différentes instances d'associer des informations d'état à une requête (par exemple pour déterminer le temps de réponse associé) et d'employer des optimisations de type *piggybacking*.

Génération de requêtes Dans le scénario le plus simple, une requête traverse toutes les instances de composants qui constituent un chemin. Il peut cependant y avoir des cas plus complexes. Une instance de composant implémentant une fonctionnalité de type RAID reçoit une requête en entrée et doit, en conséquence, en générer plusieurs en sortie. Dans le sens contraire, ces différentes requêtes sont en quelque sorte « fusionnées » pour ne produire qu'une seule réponse. En d'autres termes, il n'y a pas nécessairement de règle de « conservation des requêtes » au sein d'un chemin. Par ailleurs, même si une large majorité des requêtes correspondent à des ordres de lecture ou d'écriture émis par le client et traversent l'intégralité d'un chemin, il est possible que certaines requêtes soient « spontanément générées » au milieu d'un chemin. Par exemple, une instance implémentant un cache *write-back* produit de manière périodique des requêtes d'écriture sur disque.

4.2.2 Granularité d'un composant

Notre proposition est de ce point de vue assez conservatrice par rapport à l'existant (systèmes de fichiers empilables, Swarm, Abacus). Une extension implémente généralement un service de granularité moyenne : cache, schéma de répartition (RAID), protocole de transport réseau, partage d'accès à un organe de stockage, etc. Un tel de degré de découpage nous paraît raisonnable car il permet à la fois d'établir une distinction claire entre le rôle des différents étages d'une chaîne de traitement, de réutiliser bon nombre de composants dans divers assemblages et de limiter la structure d'un service global à un petit nombre d'étages (moins de 10, voire moins de 5 étages).

4.2.3 Interfaces d'un composant

Les caractéristiques essentielles de l'interface d'un composant Proboscis peuvent être résumées par les quelques points ci-dessous. Ces considérations sont valables pour n'importe quel *type* de composant (cf. 4.2.4).

- Un composant présente deux *faces*[3] : l'une est orientée vers la *racine* du chemin (c'est à dire vers le client qui cherche à accéder aux données), l'autre vers la *source* de données (le ou les serveurs de stockage).
- Sur chaque face, un composant présente un certain nombre de *ports* (les différentes configurations sont détaillées en 4.2.4).
- Pour une face donnée, tous les ports offrent le même *type d'interface*.
- Un type d'interface est défini par deux critères :
  - un *niveau d'interface*, désigné par un nom (par exemple *fichier*, *bloc*, *objet* ou encore *log*),

[3] *root side* et *tip side* en anglais

  - l'ensemble des méthodes fournies par le composant[4].

- Les méthodes sont toujours au nombre de deux (`issue_fn` et `done_fn`) et ont toujours la même signature quel que soit le niveau d'interface. Cet aspect est détaillé en 4.3.1.2.
- Un port ne peut (à un instant donné) être lié qu'à un seul et unique port d'un autre composant. De plus, un port placé sur une face racine ne peut être lié qu'à un port placé sur une face source et inversement. La notion de liaison est détaillée en 4.2.6.

Remarque : En l'absence de précisions, lorsque nous utilisons des termes tels que *amont/aval* ou *avant/après* pour décrire les positions relatives de composants au sein d'un chemin, ces descriptions sont implicitement établies par rapport à la direction *source*.

4.2.4 Types de composants

Le type d'un composant est défini par trois paramètres : nombre de ports sur chaque face, niveau d'interface de chaque face, nom unique. Les vérifications de typage préalables à la liaison de deux composants sont faites par rapport au niveau d'interface associé aux faces des composants concernés et aux ports disponibles. De plus, un port associé à la face source d'un composant ne peut être lié qu'à un port associé à la face racine d'un autre composant et n'étant pas déjà lié.

On peut regrouper les différents types de composants définissables au sein de Proboscis en fonction du nombre de ports qu'ils exportent sur chaque face. Deux précisions s'imposent avant de débuter notre classification :

- Tout composant Proboscis doit avoir sur au moins l'une de ses faces, un nombre de ports strictement égal à un. Cette règle correspond à la structuration d'un chemin sous forme de pipeline arborescent. Ceci assure aussi l'existence, pour chaque composant d'un point de concentration et permet de simplifier les mécanismes de reconfiguration de manière significative, sans vraiment restreindre les possibilités offertes par le modèle[5].
- Dans le cas des composants ayant de multiples ports sur une face, il est possible que certains d'entre eux ne soient pas liés sans que cela pose problème. Par exemple, une extension dont la fonction est de multiplexer l'accès à un chemin menant à un organe de stockage peut voir son nombre de clients (et donc de liaisons sur les ports de sa face racine) varier au cours du temps. Afin d'optimiser la gestion des ressources utilisées et de représenter finement à l'exécution l'évolution des liaisons entre instances, les ports d'une instance de composant peuvent être créés et détruits de manière dynamique. En conséquence, la définition d'un type de composant n'inclut pas un nombre définitif mais plutôt le nombre maximum de ports qu'il est autorisé à créer sur chacune de ses faces.

[4]Plus précisément, notre modèle de composition présente à la fois des ressemblances et des divergences avec la notion de services requis/fournis. Le niveau d'interface associé à la face racine (respectivement source) d'un composant peut s'apparenter à un service requis (resp. fourni) et chaque composant est associé à une fonction précise. Cependant, chaque port d'un composant n'est pas associé à un service différent et chaque composant peut aussi être vu comme un morceau de chemin, un « passage obligé » pour toutes les requêtes qui traversent une pile de protocoles.

[5]Nous n'avons, en effet, pas trouvé d'exemple justificatif de l'absolue nécessité d'utiliser des composants avec de multiples ports sur les deux faces. Dans certains domaines applicatifs tels que celui de Click (canevas pour la construction de systèmes de routage, cf. 4.6), il apparaît naturel de recourir à de tels éléments de base car ils correspondent parfaitement à certaines fonctionnalités attendues d'un routeur de paquets (et ceci permet aussi de limiter le nombre total d'instances de composant déployées, aspect critique en termes de performances car une configuration typique bâtie avec Click regroupe plusieurs dizaines d'instances de composants). Dans le contexte des systèmes de stockage, ce besoin est nettement moins évident : (1) les fonctionnalités de routage sont souvent bien moins raffinées, (2) et l'on peut généralement décomposer une fonction ayant plusieurs entrées et plusieurs sorties en deux sous-fonctions séquentielles pour contourner la restriction du modèle.

Compte tenu des règles du modèle de composition définies ci-dessus, on peut définir six catégories d'extensions Proboscis à partir de leur « géométrie », elles-mêmes regroupables en trois grandes familles, qui font l'objet des sections 4.2.4.1 à 4.2.4.3. Ces différents types d'extensions sont représentés sur la figure 4.1. Par la suite, les ports racine et source ne seront pas systématiquement sur les schémas.

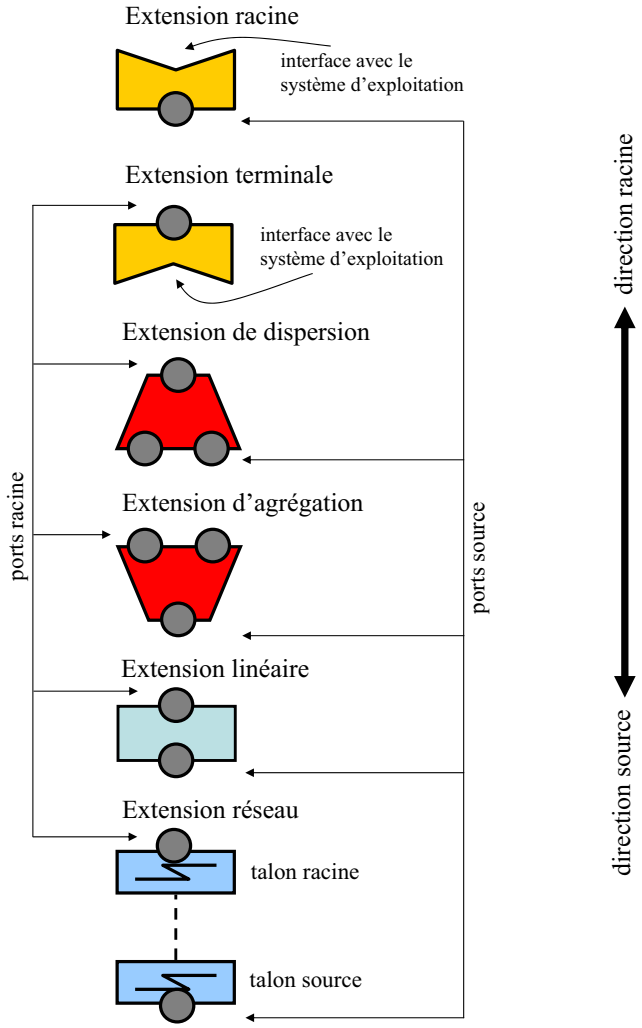


Figure 4.1 – Représentation des différents types d'extensions du canevas Proboscis

4.2.4.1 Extensions d'extrémité

On peut distinguer deux types d'extensions utilisées à l'extrémité d'un chemin : les *extensions racines* et les *extensions terminales*.

Extensions racines Une extension racine correspond à l'extrémité supérieure d'un chemin, elle ne possède aucun port racine et uniquement un port source. A l'exception de certaines extensions de

test ayant pour mission l'injection automatisée de charge, une extension racine sert généralement d'interface entre la pile de pilotes/couches d'E/S du système d'exploitation d'un nœud client et l'infrastructure Proboscis. Les requêtes émanant du système d'exploitation sont converties en requêtes internes à notre canevas (voir 4.3.1), qu'il s'agisse de requêtes d'administration (création, destruction, reconfiguration d'un chemin) ou de requêtes d'E/S. Une extension racine peut être « reliée » au système d'exploitation hôte à un niveau d'interface variable : fichier, objet, bloc, log, etc.

Extensions terminales Les extensions terminales jouent un rôle symétrique par rapport à celui des extensions racines ; elles possèdent un port racine et aucun port source. Comme son nom l'indique, une extension terminale délimite la fin d'un chemin géré par notre infrastructure. Certaines extensions terminales de test ne servent qu'à acquitter les requêtes. Dans un cas plus général (et utile), une extension terminale a pour rôle de répondre aux requêtes d'E/S émises le long d'un chemin. Une telle extension peut soit encapsuler le médium de stockage (cas d'un ramdisk géré par le canevas) soit servir d'interface avec les couches basses du système d'exploitation du nœud serveur concerné (cas d'un disque dur, d'un SGF ou d'un ramdisk gérés par le système d'exploitation du nœud serveur).

4.2.4.2 Extensions asymétriques

Une extension asymétrique présente un seul port sur l'une de ses faces et plusieurs ports sur l'autre. Nous en définissons deux types : les extensions de dispersion (un seul port racine, un nombre m ($>$1) de ports source) et celles *d'agrégation* (un nombre m ($>$1) de ports racine, un seul port source).

Extensions de dispersion Une extension de dispersion est principalement utilisée pour implémenter un schéma de répartition de données : striping, duplication, redondance par codes correcteurs d'erreurs (etc.) pour un niveau d'interface donné : fichier, bloc (RAID), etc. Lorsqu'elle reçoit une requête sur son port racine, une telle extension de dispersion émet une (voire plusieurs) requête(s) modifiée(s) sur un sous-ensemble n de ses ports sources ($1 \leq n \leq m$). Cette extension attend l'acquittement des n requêtes filles pour acquitter la requête mère (éventuellement mise à jour, par exemple dans le cas d'une requête de lecture).

Une extension de dispersion peut aussi être associée à une fonction de routage (non liée à un schéma de répartition des données) et orienter ou diffuser des requêtes (principalement d'administration — par exemple des requêtes d'invalidation de caches) vers un ensemble de sous-chemins (potentiellement réduit à un singleton).

Dans certains cas, une extension de dispersion doit gérer plusieurs flux de données. Ainsi, lorsqu'une extension RAID 5 reçoit une requête d'écriture sur son port racine, elle doit commencer par lire les informations de parité correspondantes sur les périphériques de stockage, avant de leur envoyer les ordres d'écriture. L'extension `copy` (présentée en 6.5.2) doit, quant à elle, d'une part transférer toute requête d'écriture (reçue sur son port racine) sur l'ensemble de ses ports source et, d'autre part, lire les données depuis son premier port source et les envoyer sur son second port source.

Extensions d'agrégation Ce type d'extension permet de faire converger plusieurs chemins vers un chemin unique (dans la direction *source*). Dans la plupart des configurations, une extension d'agrégation est placée juste en amont (dans la direction source) d'une extension terminale et permet de multiplexer l'accès au médium de stockage encapsulé par cette dernière. A ce titre, une extension d'agrégation peut implanter une politique d'ordonnancement de requêtes (potentiellement complexe, basée sur la provenance et/ou le contenu des requêtes, etc.).

4.2.4.3 Extensions d'interposition

Une extension d'interposition possède exactement un port d'entrée et un port de sortie. Comme les précédents, ce type englobe deux sous-catégories : les *extensions linéaires* et les *extensions réseau*.

Extensions linéaires La notion d'extension linéaire englobe la quasi-totalité des extensions d'interposition. Une extension de supervision peut fournir des types de services variés : supervision (prélèvement de statistiques pour vérifier le bon fonctionnement ou les bonnes performances du système, cryptage de données, contrôle d'accès, traduction entre deux niveaux d'interface distincts (par exemple, un SGF sert de traducteur entre une interface fichiers et une interface blocs), etc.

Extensions réseau Elles correspondent à un service bien particulier : le transport des requêtes et des données sur le réseau, de façon transparente pour les autres extensions qui composent un chemin. Chaque extension réseau est associée à un type d'interface réseau précis (par exemple une socket TCP/IP, l'interface fournie par le pilote SCI de bas niveau, VIA, Myrinet-GM, etc.) et se décompose en deux éléments : un *talon racine* (présentant un port racine sur le nœud « client ») et un *talon source* (présentant un port source sur le nœud « serveur »). Du point de vue des autres types d'extensions, un extension réseau apparaît comme une extension linéaire. Ainsi, une connexion réseau n'est pas représentée par une liaison entre composants mais par un composant à part entière. Les mécanismes flexibles de transferts de données et les détails d'implémentation d'une extension réseau sont décrits en détails au chapitre 5.

Une extension réseau peut être basée sur n'importe quelle interface/protocole de communication. Cependant, notre modèle exige que toute extension réseau fournisse des garanties de fiabilité et d'ordre pour les échanges de messages. Ainsi, un programmeur développant une extension réseau basée sur des sockets UDP doit lui-même ajouter les mécanismes nécessaires pour garantir la transmission fiable et ordonnée des messages. Cette contrainte supplémentaire est bien en phase avec la sémantique généralement associée aux systèmes de stockage et permet de confiner la gestion des problèmes de transmission au sein des extensions réseau, sans impact sur les autres composants d'un chemin.

Contrairement à la règle habituelle, les ports d'une extension réseau ne sont pas typés. Ceci permet d'utiliser le même composant pour assurer le transport de requêtes à différents niveaux d'interface. Les composants reliés de part et d'autre à une extension réseau doivent cependant présenter le même type d'interface sur la face concernée.

4.2.4.4 Autre axe de classification

On peut également trier les différents types de composants gérés par notre canevas en fonction du traitement qu'ils appliquent aux requêtes qu'ils reçoivent, à la fois au niveau du routage et à celui du contenu de la requête. À ce sujet, le comportement d'une extension peut varier en fonction du type de requêtes considéré. Puisque le principal service fourni par les chemins construits grâce à notre canevas correspond à l'accès à un support de stockage, cette seconde classification se base sur les effets associés aux requêtes de lecture et d'écriture de données.

Aucun effet sur le routage ni le contenu Une extension linéaire de supervision (*monitoring*) ne fait qu'examiner les requêtes qui la traversent pour prélever des statistiques. De façon similaire, une extension réseau transfère les requêtes d'un nœud à un autre sans y apporter la moindre modification. Il en va de même pour une simple extension d'agrégation qui multiplexe l'accès à un support de stockage : elle peut mettre en œuvre une politique d'ordonnancement particulière mais ne modifie pas

le contenu des requêtes ni leur routage : à « l'aller », une requête d'accès aux données est propagée sur l'unique port source disponible ; dans le sens du « retour », la requête est acquittée sur son port racine d'origine.

Nous englobons également les extensions d'extrémité dans cette catégorie. Une extension racine convertit des requêtes du système d'exploitation en requêtes internes à l'infrastructure Proboscis. Elle constitue en cela le point de départ d'un chemin et crée les requêtes internes appropriées (nous effectuons une distinction entre la création d'une requête Proboscis et la modification d'une requête existante). Une extension terminale répond aux requêtes d'accès aux données et l'on pourrait par exemple considérer que répondre à une requête de lecture implique de la modifier (pour y adjoindre les données requises). Nous ne souscrivons pas à cette vision des choses ; notre choix à ce sujet sera éclairci au chapitre 5, qui détaille les mécanismes de transfert de données.

Effet sur le routage uniquement Une extension de dispersion assurant uniquement un rôle de routage appartient évidemment à cette catégorie. Une extension linéaire de contrôle d'accès peut refuser de propager une requête et l'acquitter « prématurément » (c'est-à-dire avant qu'elle n'ait atteint sa destination prévue). Une extension de dispersion implémentant un schéma de redondance par duplication (complète) est également assimilable à cette classe de composants. Dans ce cas, pour une requête d'écriture, il y a création de requêtes filles vers chacun des sous-chemins concernés par la duplication des données (comme expliqué en 4.2.4.2) mais le contenu de la requête n'est pas modifié[6]. Nous assimilons cette création de requêtes filles à une opération de routage.

Effet sur le contenu uniquement Certaines extensions linéaires agissent sur le contenu d'une requête. Ainsi, une extension de cryptage encode les données écrites et décode les données lues. Il peut également s'agir d'une extension jouant un rôle de translation de niveau d'interface. Par exemple, une extension implémentant un SGF — ou du moins la couche de gestion des données d'un SGF (cf 2.1.2.5 et 2.2)— modifie le niveau d'abstraction d'une requête d'accès aux données (inode sur sa face racine, bloc sur sa face source).

Dans le premier exemple (cryptage), l'adressage des données n'est pas modifié (on conserve les mêmes indicateurs inode + décalage ou numéros de blocs, selon le niveau d'interface). Dans le second exemple, c'est le contraire : les informations à écrire ne sont pas modifiées mais les addresses de destination le sont.

Les effets peuvent également porter sur la taille et la valeur des données, dans le cas d'une extension qui englobe les fonctions de compression (des données écrites) et de décompression (des données lues). Une telle extension peut être utilisée seule (pour effectivement compresser les données à stocker) et être placée de façon interchangeable au niveau du client ou du serveur (selon le nœud le mieux adapté pour endosser la charge associée à cette tâche) ou en paire (l'une sur le client et l'autre sur le serveur), pour diminuer le volume de données circulant sur le réseau.

Effet sur le routage et le contenu La quasi-totalité des extensions de dispersion implémentant un schéma de répartition des données[7] (*striping*, redondance à base de codes correcteurs) peuvent modifier à la fois le routage et le contenu de certaines requêtes. Prenons l'exemple, d'une extension implémentant une schéma RAID 5 avec quatre supports de stockage distincts et une taille de morceau (*chunk size*) égale à 8 fois la taille de bloc qu'elle exporte. Une requête de lecture portant sur les blocs

[6] Une extension de routage qui transmet une requête vers plusieurs sous-chemins utilise également des requêtes filles dont le contenu est identique à celui de la requête mère.

[7] à l'exception de la duplication complète.

0 à 31 va engendrer le génération de plusieurs requêtes filles avec une modification d'adressage des données par rapport à celle-ci :

- pour le sous chemin associé au premier support de stockage, requête de lecture pour les blocs 0-15 (correspondant aux blocs 0-7 et 24-31 demandés par la requête initiale),
- pour le sous chemin associé au deuxième support de stockage, requête de lecture pour les blocs 0-7 (correspondant aux blocs 8-15 demandés par la requête initiale),
- pour le sous chemin associé au troisième support de stockage, requête de lecture pour les blocs 0-7 (correspondant aux blocs 16-23 demandés par la requête initiale),
- pour le sous chemin associé au quatrième support de stockage, pas de requête (car le support 4 stocke le chunk de parité de la première bande de données, le support 3 stocke le chunk de parité pour la seconde bande, etc.).

Cet exemple illustre bien la modification du contenu d'une requête (au niveau de l'adressage des données) combinée à une action de routage (génération sélective de commandes filles).

Les extensions qui modifient le contenu (et plus précisément l'adressage des données) d'une requête doivent enregistrer (auprès de l'infrastructure Proboscis) une méthode permettant d'effectuer la conversion d'adressage induite par leur sémantique (dans chaque sens).

Ainsi, l'infrastructure d'exécution peut chaîner les méthodes de conversion d'adressage en fonction de la composition du chemin considéré, afin de pouvoir comparer différents niveaux d'adressage : il est possible de déterminer si deux adresses utilisées à différents niveaux d'un chemin correspondent à la même donnée. Ceci rend possible l'utilisation d'un cache de données global, commun à toutes les instances au sein d'un nœud, même en présence d'extensions qui modifient l'adressage des données.

4.2.5 Règles complémentaires pour la programmation d'extensions

Maintenant que le modèle de composants a été présenté de manière plus détaillée, nous pouvons poursuivre de façon plus précise la discussion entamée en 4.2.2 sur la granularité et la nature des fonctions associée à une extension. Afin de limiter la complexité et de clarifier le rôle de chaque extension au sein d'un chemin, notre modèle définit également les quelques règles de structuration suivantes.

- Une extension qui opère une translation de niveau d'interface (c'est-à-dire qui ne présente pas le même niveau d'interface sur ses deux faces, par exemple fichier côté racine et bloc côté source) ne doit pas avoir d'autres implications sur le routage et la transformation des requêtes. Ceci exclut d'employer une extension asymétrique à cet effet. Pour les mêmes raisons, il n'est pas satisfaisant d'utiliser une extension d'extrémité, dont le rôle premier est de convertir des requêtes propres à Proboscis en requêtes destinées au système d'exploitation hôte (et vice versa). Ainsi, seules les extensions linéaires sont adaptées pour fournir des fonctions de translation d'interface.
- Une extension d'agrégation a pour rôle de multiplexer les requêtes de différents clients vers un chemin unique, selon une certaine politique d'ordonnancement. Elle n'est pas en charge d'autres tâches susceptibles de modifier le routage ou le contenu des requêtes (contrôle d'accès, cryptage, etc.). Ces fonctions doivent être associées à d'autres extensions, en aval de l'extension d'agrégation.

Ces règles de programmation ont pour vocation d'inciter le développement d'extensions réutilisables au sein de différentes configurations, simples à maintenir et à reconfigurer en limitant et clarifiant au maximum la fonction de chacune d'elles. En contrepartie, ces règles imposent un léger surcoût.

- Pour une fonctionnalité globale équivalente, un chemin basé sur ces principes nécessitera quelques extensions de plus qu'une configuration optimisée, où différentes fonctions proches sont groupées au sein d'une même extension.
- En outre, certaines fonctions peuvent être retardées. Par exemple, dans le cas du filtrage d'accès, une requête incorrecte arrivant sur un nœud serveur sera détectée plus tardivement si les vérifications ne sont pas effectuées par l'extension d'agrégation. Ainsi, certaines ressources seront gaspillées pour acheminer une requête « inutile » un peu plus loin au sein du chemin alors qu'elle aurait pu être détectée plus tôt.

Nous pensons que ces désagréments sont relativement mineurs. D'une part, même avec le souci d'isoler précisément chaque fonction au sein d'une extension distincte, la taille d'un chemin reste limitée. D'autre part, nous nous intéressons à un contexte où les problèmes de sécurité ne sont pas primordiaux. Les serveurs de stockage sont déployés dans un environnement réseau sécurisé et se situent à le fin de la « chaîne applicative ». En conséquence, une large majorité des requêtes d'accès aux données sont légitimes. Ainsi, nous ne subissons pas dans les mêmes proportions les contraintes qui pèsent sur les serveurs web, dont les piles de protocoles réseau doivent opérer une classification anticipée et agressive des paquets pour éviter la surcharge due à une attaque par déni de service.

4.2.6 Liaison

Deux instances de composants déployées sur un même nœud (et occupant une position adjacente au sein d'un chemin) sont liées par une « liaison langage ». La structure en mémoire représentant une instance déployée contient, pour chaque port utilisé sur chaque face, un pointeur vers le descripteur de l'instance concernée.

En revanche, comme indiqué en 4.2.4.3, la liaison entre deux instances déployées sur deux nœuds distincts est représentée de façon explicite, par une extension réseau. Cette différence de représentation s'explique par les raisons suivantes.

- Dans notre contexte, il n'est pas nécessaire de permettre une large palette de modes d'interactions entre deux instances déployées sur un même nœud. En conséquence, la gestion des liaisons locales est confiée à l'infrastructure d'exécution (code générique pour toutes les extensions) et l'objectif premier est d'optimiser leur coût, à la fois en termes de temps d'exécution (établissement, libération et surtout traversée) et de consommation mémoire.
- Les préoccupations principales sont différentes dans le cas d'une liaison distante. Contrairement aux communications locales qui ne nécessitent pas de copie de données, les interactions entre deux extensions distantes ont un impact majeur sur les performances globales d'un chemin. Pour permettre de configurer de manière optimale (et éventuellement d'adapter dynamiquement) les propriétés de communication associées à un chemin, il nous faut disposer d'une représentation explicite du canal de communication.

Les mécanismes d'établissement et de destruction d'une liaison sont détaillés dans la section 4.4.2, relative au déploiement. L'utilisation d'une liaison pour l'acheminement de requêtes est évoquée lors de la présentation du modèle d'exécution (4.3).

4.3 Modèle d'exécution

Jusqu'à présent, nous n'avons pas explicité la façon dont sont programmées les extensions qui composent un chemin ni de quelle manière le code correspondant est exécuté. Nous nous sommes contenté d'évoquer la notion de « requêtes » traversant un chemin. Cette section commence par préciser le modèle imposé pour la programmation d'un composant Proboscis. Elle détaille ensuite les

mécanismes de base permettant à différents composants d'interagir ainsi que les modèles de communication qui en découlent, puis s'achève par la description de l'infrastructure d'exécution de Proboscis.

4.3.1 Programmation d'un composant

Afin d'imposer le moins de restrictions possibles sur le flux de contrôle et de données qui traverse un chemin, les extensions Proboscis interagissent de manière asynchrone. Ce modèle d'interaction procure plusieurs avantages significatifs. Tout d'abord, il est bien adapté à la nature asynchrone de certaines opérations d'E/S et de transferts de données à distance. De plus, il facilite la mise en œuvre des opérations de reconfiguration dynamique.

Plus précisément, chaque extension est implémentée sous la forme d'une machine à états qui réagit à l'arrivée d'événements. Nous distinguons deux catégories principales d'événements : d'une part, l'arrivée d'une requête sur un port et, d'autre part, tous les autres types d'événements. Nous commençons, en 4.3.1.1, par décrire comment sont gérés ces événements puis nous présentons l'interface de programmation correspondante en 4.3.1.2.

4.3.1.1 Commandes et événements

Commandes Compte tenu des précisions qui viennent d'être apportées, les requêtes entre composants Proboscis prennent la forme de messages typés, appelés *commandes*. Dorénavant, nous utiliserons ces deux termes (*requête* et *commande*) de manière interchangeable.

Lorsqu'une commande est émise sur le port d'une extension, ceci déclenche l'exécution d'un traitant approprié, déterminé en fonction du type de la commande. De manière générale, un traitant examine les paramètres de la commande, décide en conséquence d'effectuer certaines actions (y compris de franchir une éventuelle transition vers un autre état de l'automate), puis détermine le routage à appliquer à la commande (ce qui peut induire la génération de commandes filles, comme expliqué en 4.2.4).

Au sein d'un même nœud, une commande est transmise par un simple passage de pointeur. Dans le cas d'un chemin réparti, lorsqu'une commande atteint une extension réseau, elle est sérialisée et recopiée sur le nœud partenaire. Les problèmes potentiels de concurrence d'accès ou de divergence de duplicas répartis pour les commandes sont évités de manière simple, grâce aux principes de base d'un chemin.

- Seule l'instance qui est couramment en charge d'une commande (l'instance au sein de laquelle la commande a été émise et pas encore propagée ou acquittée) est autorisée à la manipuler et *a fortiori* à accéder à son contenu.
- Lors du trajet retour d'une commande (acquittement), les éventuelles modifications apportées au contenu d'une commande sont propagées par l'extension réseau (ou les extensions réseau concernées) vers le(s) nœud(s) hébergeant la racine d'un chemin.

De nombreuses instances peuvent avoir besoin d'associer des informations à une commande et de les retrouver lorsque la même commande acquittée les traverse à nouveau. Il peut, par exemple, s'agir d'informations de supervision (estampilles temporelles pour évaluer la latence d'une requête) ou de routage (pour permettre à une extension d'agrégation de retrouver sur quel port de sa face racine elle doit acquitter une commande), etc. Dans tous les cas, il est nécessaire de fournir un mécanisme simple et efficace pour associer un état (spécifique à une instance) à une commande. Pour ce faire, chaque commande possède une pile qui permet de conserver les informations nécessaires entre l'aller et le retour d'une commande. La pile étant implémentée de façon globale (une seule pile par commande

pour toutes les instances déployées sur un même nœud), chaque instance a pour responsabilité de dépiler les informations qui lui sont propres avant d'acquitter une commande.

Par ailleurs, différentes instances déployées au sein d'un même chemin peuvent dans certains cas avoir besoin d'échanger des informations d'état associées à une commande (distinctes des paramètres de la commande). Par exemple, une extension racine peut vouloir indiquer à une extension d'agrégation (située plus loin au sein du chemin) quelle est la classe de service requise pour une commande donnée. À cet effet, une instance peut annoter une commande en lui ajoutant une annotation, que d'autres instances pourront consulter et éventuellement modifier.

Certaines commandes d'administration peuvent être définies comme étant bloquantes. Lorsqu'une commande bloquante traverse une instance, le port par lequel elle est entrée est désactivé. Toutes les commandes arrivant sur ce port sont alors mises en attente jusqu'à ce que la commande bloquante traverse à nouveau le port en sens inverse.

Autres types d'événements En l'absence de précision, nous employons le mot *événement* pour désigner tout type d'événement qui ne correspond pas à l'arrivée d'une commande sur le port d'une extension. Contrairement à une commande, un événement n'est pas transmis d'une instance à une autre, il est confiné au sein d'une instance précise. L'occurrence d'un événement déclenche également l'exécution d'un traitant adapté (basé sur le type de l'événement considéré). Un événement peut être associé à un ensemble de paramètres. Ces derniers n'influent pas sur la nature de la réaction produite par le traitant mais lui fournissent les informations nécessaires à son bon déroulement. Les événements jouent deux rôles principaux pour les composants Proboscis :

- la planification d'activités périodiques : par exemple, la synchronisation sur disque du contenu d'un cache ;
- l'interface avec des événements extérieurs au canevas : arrivée d'une requête d'E/S à l'entrée d'un chemin, notification de terminaison d'un transfert réseau ou disque, etc.

Dans les deux cas, un traitant d'événement sert de point de départ à une chaîne d'interactions : il entraîne, dans la plupart des cas, l'émission d'une commande (génération, acquittement, ou propagation d'une commande bloquée) qui va successivement traverser les différentes instances qui composent le chemin concerné.

Définition de commandes et d'événements Proboscis ne définit à la base qu'un ensemble minimal de commandes (création, destruction, modification de la structure d'un chemin). Tous les autres types de commandes doivent être définis par les extensions elles-mêmes. L'infrastructure fournit l'interface nécessaire pour permettre à une extension d'enregistrer de nouveaux types de commandes lors de son propre enregistrement. Ainsi un ensemble d'extensions coopératives (par exemple pour implémenter les fonctionnalités associées à un niveau d'interface particulier tel que SCSI ou OSD) peut définir les commandes qui lui sont nécessaires. Pour enregistrer un nouveau type de commande, une extension doit spécifier un ensemble de fonctions associées :

- des fonctions de sérialisation/désérialisation (transformation d'une commande en une suite d'octets et vice versa) utilisées par les extensions réseau pour la transmission des commandes d'un nœud à un autre ;
- une fonction permettant d'identifier les paramètres d'une commande qui correspondent à des addresses de données stockées (et le niveau d'interface concerné, par exemple bloc), nécessaire pour les extensions susceptibles de consulter et/ou modifier l'adressage des données référencées par une commande ;

- une fonction permettant d'identifier les paramètres d'une commande qui correspondent à des tampons d'E/S (voir chapitre suivant).

Si la commande a déjà été enregistrée (par une autre extension), avec un profil strictement identique alors la commande n'est pas réenregistrée mais un compteur d'enregistrement est incrémenté (de façon à ne supprimer l'enregistrement que lorsqu'il n'y aura plus aucune extension enregistrée utilisant ce type de commande). Si la commande a déjà été enregistrée (même nom) avec un profil différent, alors le nouvel enregistrement échoue.

La définition d'un type d'événement n'est valable qu'au sein d'un type d'extension donné. Les différents types d'événements nécessaires au fonctionnement d'une extension sont définis dans le code de celle-ci. Il n'y a pas besoin d'enregistrer un type d'événement auprès de l'infrastructure car différentes extensions ne peuvent pas interagir par le biais d'événements.

4.3.1.2 Interface de programmation

Nous présentons ici un aperçu de la manière dont sont programmés les composants Proboscis. Nous décrivons à la fois l'interface fournie par chaque composant et l'interface fournie par le canevas pour simplifier l'écriture d'un composant.

Méthodes d'interface d'un composant Le code d'une extension doit implémenter trois méthodes abstraites : `issue_fn`, `done_fn` et `event_fn`[8]. Nous détaillons ci-dessous le rôle de chacune d'elles.

`issue_fn` Cette méthode est appelée lors de l'arrivée d'une commande (dans la phase « aller » de son parcours) sur l'un des ports d'une instance. Les deux premiers paramètres désignent respectivement l'instance et le descripteur de la commande concernée. Le troisième paramètre indique la direction (source ou racine[9]) dans laquelle est émise la commande et, par incidence, sur quelle face de l'instance elle arrive (une commande émise dans la direction source arrive sur la face racine d'une instance et inversement). En général, une commande est initialement émise dans la direction de la source de données, puis, une fois acquittée, parcourt le chemin en sens inverse (en direction de la racine du chemin). Certaines commandes particulières (principalement liées à l'administration et à la reconfiguration), générées par une instance au milieu ou à une extrémité (source) d'un chemin, peuvent cependant effectuer un trajet contraire : émission en direction racine et acquittement en direction source. Le paramètre `dir` permet ainsi de préciser quelle est la face de l'instance concernée par l'arrivée de la commande. Enfin, le dernier paramètre indique sur quel port est émise la commande et la valeur de retour permet de signaler une éventuelle erreur de traitement.

La méthode `issue_fn` sert de point d'entrée pour le traitement de commandes en phase « aller ». Son rôle principal consiste à appeler le traitant approprié en fonction du type de la commande reçue. Ces traitants spécialisés sont internes au composant et n'ont pas à être exportés à l'extérieur.

`done_fn` Cette méthode joue un rôle semblable à `issue_fn` pour l'arrivée d'une commande (en phase de « retour », d'acquittement). Dans la plupart des cas, une commande acquittée arrive sur la face source d'une instance (et donc en direction racine).

[8] Pour une extension donnée, les nom des méthodes est préfixé par le nom de l'extension.
[9] respectivement *tip direction* et *root direction* en anglais

`event_fn` Cette méthode est appelée lors de l'occurrence d'un événement au sein d'une instance. Le premier paramètre identifie l'instance concernée, le second précise le type de l'événement, le dernier est un pointeur vers d'éventuels paramètres associés à l'événement. Comme les deux précédentes méthodes, `event_fn` sert de point d'entrée pour le traitement d'un événement et a pour rôle principal d'appeler un traitant spécialisé en fonction du type d'événement considéré.

En complément des trois points d'entrée que nous venons d'évoquer, le programmeur d'une extension doit définir un ensemble de traitants spécifiques à un type de commande ou d'événement. Pour un type de commande donné, deux traitants sont généralement nécessaires, un pour chaque sens : émission et acquittement (appelés respectivement par `issue_fn` et `done_fn`). Un type d'événement n'est pas associé à un parcours bidirectionnel et ne nécessite par conséquent qu'un seul traitant.

Une extension doit également prévoir un traitant par défaut (en fait, un traitant par direction) pour les types de commandes qu'elle ne gère pas. Ces traitants se contentent d'effectuer une opération de routage simple (il suffit souvent de transférer la commande vers l'instance suivante, dans la même direction). Dans certains cas (par exemple si l'instance courante possède plusieurs ports sur la face à partir de laquelle elle doit propager la commande), la décision de routage par défaut peut être assistée/modifiée par l'infrastructure d'exécution (voir ci-dessous).

Il n'est pas nécessaire de fournir un traitant par défaut pour les événements inconnus. En effet, les événements sont internes à une extension et résultent de son activité propre, contrairement aux commandes, qui peuvent provenir d'un autre type d'extension instanciée au sein du même chemin. Autrement dit, l'arrivée d'un événement non prévu correspond à une erreur de programmation (et peut généralement être ignorée).

Méthodes fournies par le canevas pour la communication entre instances Nous décrivons ici un petit sous-ensemble de l'interface fournie par le canevas pour simplifier l'implémentation d'une extension (et également jouer un rôle d'interposition dans certains cas). L'accent est mis sur le routage des commandes.

Au sein d'un traitant spécifique à un type de commande, il est notamment possible de :

- récupérer (et éventuellement modifier) les paramètres associés à la commande en question ;
- générer une ou plusieurs commandes (il est possible de spécifier une relation de parenté entre une commande « mère » et une ou plusieurs commande(s) « fille(s) ») ;
- stocker une commande dans une file d'attente (qui aura été préalablement créée à cet effet) ;
- transmettre la commande initialement reçue (ou des commandes générées par le traitant) le long du chemin. Il existe pour cela deux méthodes : `prob_cmd_issue` et `prob_cmd_done`. La première sert à émettre une commande (en phase « aller ») à partir du port `joint` situé sur la face `dir` de l'instance `inst`. La seconde fournit une fonctionnalité similaire pour les commandes en phase « retour ». Il est à noter que `prob_cmd_done` peut être appliqué à une commande étant déjà en phase de retour mais peut également être utilisée pour servir de point de départ à l'acquittement d'une commande (c'est à dire provoquer le « demi-tour » d'une commande).

Les fonctions d'émission `prob_cmd_issue` et `prob_cmd_done` jouent trois rôles principaux. Tout d'abord, quelle que soit la commande considérée, elles servent à déterminer quelle est l'instance destinatrice de la commande (en fonction de l'instance courante, du port et de la direction d'émission) et à invoquer la méthode appropriée (`issue_fn` ou `done_fn`) sur cette instance cible. Deuxièmement, dans le cas de commandes de création et de destruction de chemin, elles effectuent les opérations d'administration nécessaires. Par exemple, lors de la création d'un chemin, `prob_cmd_issue` et `prob_cmd_done` gèrent de manière transparente l'instanciation, l'initialisation et la liaison des

composants (voir la section 4.4.2 pour plus de détails). Enfin, pour les commandes de reconfiguration dynamiques, elles ajustent les décisions de routage prises par les extensions, ce qui permet de décharger les programmeurs d'extensions de cette tâche.

Méthodes fournies par le canevas pour la génération d'événements L'interface de programmation d'un composant Proboscis fournit deux méthodes pour planifier l'occurrence d'un événement : `prob_sched_event` et `prob_sched_delayed_event`.

La première permet de demander l'occurrence d'un événement de type event au sein de l'instance `inst` dès que possible. La seconde ajoute un délai minimum à respecter avant l'occurrence de l'événement. Nous verrons en section 4.3.3 comment (et par quelles entités) ces méthodes de gestion d'événements peuvent être appelées.

4.3.2 Modèle de communication

Cette section décrit les différentes stratégies de routage utilisables pour la propagation d'une commande au sein d'un chemin. Elle explicite ensuite les responsabilités d'une extension réseau par rapport à d'éventuels problèmes de communication entre deux composants distants.

4.3.2.1 Modes de routage

Comme nous l'avons vu plus haut, les instances qui constituent un chemin communiquent par l'envoi de commandes. Une instance émet une commande sur l'un de ses ports et reçoit ultérieurement, de manière asynchrone, un acquittement correspondant sur le même port. Une commande est propagée/routée au travers d'un chemin jusqu'à sa destination finale, où elle est acquittée ; elle parcourt alors le même chemin en sens contraire, jusqu'à son instance d'origine.

Une commande peut être véhiculée au sein d'un chemin selon différentes politiques de routage, détaillées ci-dessous et représentées en figure 4.2. La première s'applique à toutes les commandes impliquées dans la manipulation de données. Pour les autres types de commandes (commandes d'administration), plusieurs modes de communication sont possibles.

Routage en fonction de l'adressage des données Une commande d'accès à des données est routée en fonction de la plage de données qu'elle cible (par exemple, au niveau d'interface « blocs », l'intervalle des numéros de blocs visés). Ce type de routage est effectué au niveau des extensions de dispersion (cf. 4.2.4.2), grâce à la méthode (associée à la définition de la commande concernée) permettant d'identifier le(s) paramètre(s) correspondant à des adresses de données (voir 4.3.1.1).

Routage en fonction d'un identifiant d'instance cible Certaines commandes d'administration peuvent être ciblées pour une instance spécifique (chaque instance de composant Proboscis dispose d'un identifiant unique à l'échelle de la grappe). Contrairement au routage progressif pour les commandes d'accès aux données (décrit au point précédent), il y a dans ce cas un routage « à la source » : le cheminement de la commande est entièrement déterminé dès son émission. Ce type de routage est principalement employé pour des commandes de reconfiguration qui modifient la structure d'une portion précise d'un chemin (insertion ou suppression d'instances — voir 6.4).

Diffusion dans une direction donnée Une commande peut aussi être propagée vers toutes les instances d'extensions (et donc à travers tous les sous-chemins) qui composent un chemin. Ceci implique bien sûr la création de commandes filles en chaque « point d'éclatement » (c'est-à-dire

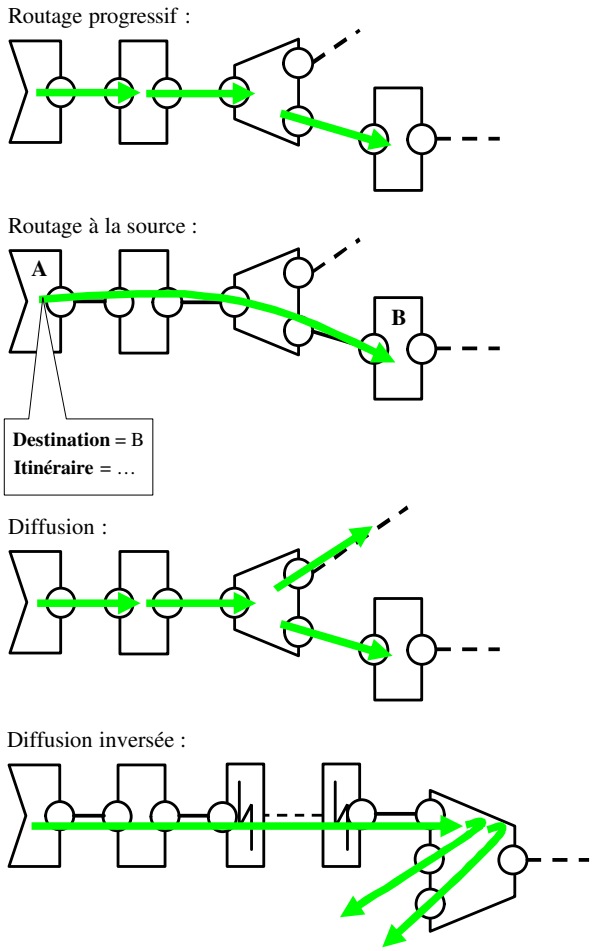


Figure 4.2 – Représentation des différents modes de routage possibles pour une commande

pour chaque extension de dispersion ou d'aggrégation, selon le sens initial de la commande) du chemin. Ce mode de routage est notamment utilisé pour les commandes de création et de destruction d'un chemin. Il s'apparente également au routage d'une commande d'écriture au sein d'une configuration basée sur la duplication complète des données (*mirroring*).

Diffusion inversée Le routage ciblé et le routage par diffusion que nous venons de décrire peuvent être combinés pour former un autre mode de routage, la diffusion inversée, articulé selon deux étapes. Tout d'abord, une commande ciblée est envoyée à une instance d'extension asymétrique. Lorsque l'instance concernée reçoit la commande ciblée, elle la propage (ou propage un autre type de commande qui lui est associé) sur tous les autres ports de la face sur laquelle la commande initiale a été reçue (et donc dans la direction opposée à celle de la commande initiale). Ce mode de communication est utile pour implémenter des protocoles de reconfiguration. Il permet, par exemple, au client d'un support de stockage partagé de contacter tous les autres clients de ce support ou encore à un organe de stockage de contacter tous les autres organes de stockage qui sont regroupés au sein du même volume logique (en utilisant respectivement une extension d'agrégation et une extension de dispersion comme routeur de diffusion).

Le routage progressif des commandes adressant des données est effectué par les extensions, tout comme la diffusion d'une commande sur toutes les branches d'un chemin. En revanche, le routage d'une commande ciblée est assuré par l'infrastructure d'exécution, capable d'intercepter toutes les communications entre instances.

Ces différents modes de routage seront illustrés au cours des sections suivantes, notamment en 4.4 ainsi qu'au chapitre 6.

4.3.2.2 Gestion des problèmes de transmission

L'infrastructure Proboscis s'appuie (en partie - voir compléments en 6.1) sur les extensions réseau pour détecter les problèmes de communication réseau et les pannes de nœuds. Si l'un des éléments (talon racine ou talon source cf. 4.2.4.3) d'une instance d'extension réseau perd le contact avec son pendant, il lui revient de déterminer s'il s'agit d'un problème transitoire ou d'une panne irrécupérable (liée au support de communication où à un dysfonctionnement du nœud distant)[10]. Dans le second cas, toutes les commandes en transit doivent être acquittées[11] (avec un signalement d'erreur). Si le lien avec le nœud inaccessible est finalement rétabli, et que les commandes considérées comme perdues sont finalement reçues, ces dernières sont ignorées.

4.3.3 Infrastructure d'exécution

4.3.3.1 Introduction

L'infrastructure Proboscis joue deux rôles principaux vis à vis des systèmes de stockage construits à partir de notre modèle de composition. D'une part, elle gère un ensemble de méta-données qui font référence à tous les types d'extensions et de commandes enregistrés. C'est sur cette fonction de base que repose la possibilité d'étendre le canevas (en ajoutant de nouveaux types de composants) et de modifier dynamiquement l'implémentation d'un composant. D'autre part, l'infrastructure Proboscis fournit les services et les ressources système nécessaires à l'exécution du système de stockage. Dans cette section, nous mettons l'accent sur le second point, en passant volontairement sous silence les aspects liés à la façon dont sont gérés les tampons d'E/S et les transferts de données (ces mécanismes sont détaillés au chapitre suivant).

Le fait de contrôler, au sein de l'infrastructure, les ressources d'exécution associées au système de stockage offre plusieurs avantages. Tout d'abord, centraliser les décisions d'allocation de ressources permet potentiellement de mieux contrôler et arbitrer la consommation des ressources système (à la fois entre les différents flux de requêtes et les différentes tâches du système de stockage et également entre la consommation globale associée au système de stockage et les autres activités du système d'exploitation qui l'héberge). En outre, ce contrôle centralisé peut également, sans grandes difficultés, fournir des garanties sur le modèle d'exécution qui simplifient la programmation des composants (l'infrastructure d'exécution garantit que deux événements ne peuvent être émis de façon concurrente vers la même instance[12]) et la gestion des opérations de reconfiguration.

De façon concrète, l'infrastructure d'exécution que nous proposons est basée sur un seul thread (par nœud, quel que soit le nombre de chemins traversant le nœud), chargé d'exécuter le code des

[10] Cette responsabilité est simplifiée par le fait que la plupart des réseaux d'interconnexion à hautes performances fournissent, en plus des garanties de fiabilité pour le transport des données, des mécanismes permettant à chaque nœud de surveiller l'état de ses correspondants ainsi que celui du médium de communication.

[11] Chaque « moitié » (talon racine ou source) d'une instance réseau conserve une copie des commandes qu'elle a émises vers l'autre moitié et qui ne sont pas encore été acquittés.

[12] Autrement dit, le modèle d'exécution assure que les événements destinés à une instance donnée sont sérialisés.

instances vers lesquelles sont émis des commandes ou des événements. Ce thread (`probed` - *Proboscis execution daemon*) dispose d'une file d'attente où sont insérés les événements à émettre vers les différentes instances locales.

Puisque qu'une instance de composant n'est pas associée à un contexte d'exécution qui lui est propre, il est nécessaire de définir explicitement les informations d'états qui doivent être sauvegardées entre deux invocations de ses traitants. Cette tâche revient au programmeur d'une extension mais l'infrastructure lui fournit les services nécessaires pour définir, sauvegarder, modifier et récupérer les informations d'état associées à une instance.

Enfin, précisons que l'infrastructure Proboscis n'offre aucun mécanisme générique de maintien de la cohérence des données manipulées par différentes entités (qu'il s'agisse de différents clients d'un support de stockage partagé ou d'extensions distinctes au sein d'un même chemin). Nous reviendrons sur ce point au en 5.5. Ce choix est lié à la volonté de ne pas imposer une sémantique particulière aux développeurs d'un système de stockage réparti.

4.3.3.2 Exemple dans un contexte centralisé

Pour illustrer le principe de fonctionnement qui vient d'être détaillé, commençons par un exemple simple, dans un cadre centralisé, avec la configuration représentée en 4.3. Le système considéré est composé de trois extensions (extension racine d'interface avec la couche bloc, extension de cryptage, extension terminale d'interface avec le PGB correspondant au disque utilisé) et n'effectue pas de translation de niveau d'interface (il accepte des requêtes de niveau bloc en entrée et produit des requêtes du même niveau en sortie).

L'extension racine se présente au système d'exploitation hôte comme un PGB, et fournit à ce titre une fonction permettant de soumettre une requête au périphérique. Lorsqu'un programme effectue une requête d'E/S sur ce PGB, il appelle cette fonction (`os2prob_issue_request`), qui insère la requête dans une file d'attente associée et demande l'occurrence d'un événement (de type `OS2PROB_BLOCK_REQUEST`) pour l'instance numéro 1. Après l'appel à `os2prob_issue_request`, le programme à l'initiative de la requête se bloque en attente de la réponse (dans le cas d'une opération d'E/S synchrone)[13]. Lorsque l'instance `os2prob` reçoit l'événement, elle exécute un traitant approprié qui crée une commande (interne à Proboscis) correspondant à la requête du système puis qui émet la commande sur l'unique port de l'instance (sur la face source) en appelant `prob_cmd_issue`. Cette fonction de routage de l'infrastructure appelle elle-même la méthode `issue_fn` de la seconde instance, qui se charge de crypter les données (dans le cas d'un requête d'écriture) puis appelle elle-même `prob_cmd_issue` pour transmettre la commande à la dernière instance. Cette dernière instance correspond à une extension terminale. Elle convertit la commande interne à Proboscis en une requête pour le PGB du disque concerné (en y insérant un pointeur permettant de retrouver la commande Proboscis de départ), spécifie un traitant (`prob2os_b_end_io`), qui sera appelé par le système lorsque l'opération d'E/S sur le disque sera terminée, stocke la commande Proboscis dans une file d'attente interne et émet la requête système sur le PGB du disque. Cette étape marque la fin du flot d'exécution (chaîne d'appels synchrones à partir du traitant de l'événement `OS2PROB_BLOCK_REQUEST`) qui a permis de véhiculer la commande de la racine du chemin à la source de données. Le thread

[13] Dans le cas d'une opération d'E/S asynchrone (lorsque ce mode d'E/S est supporté par le système d'exploitation), le programme appelant n'est pas bloqué (certaines implémentations utilisent cependant des threads qui se bloquent à la place du programme appelant). Dans tous les cas, ce comportement est uniquement lié aux fonctionnalités fournies par le système d'exploitation et est indépendant de celui de l'infrastructure Proboscis, qui peut donc gérer indifféremment des E/S synchrones ou asynchrones.

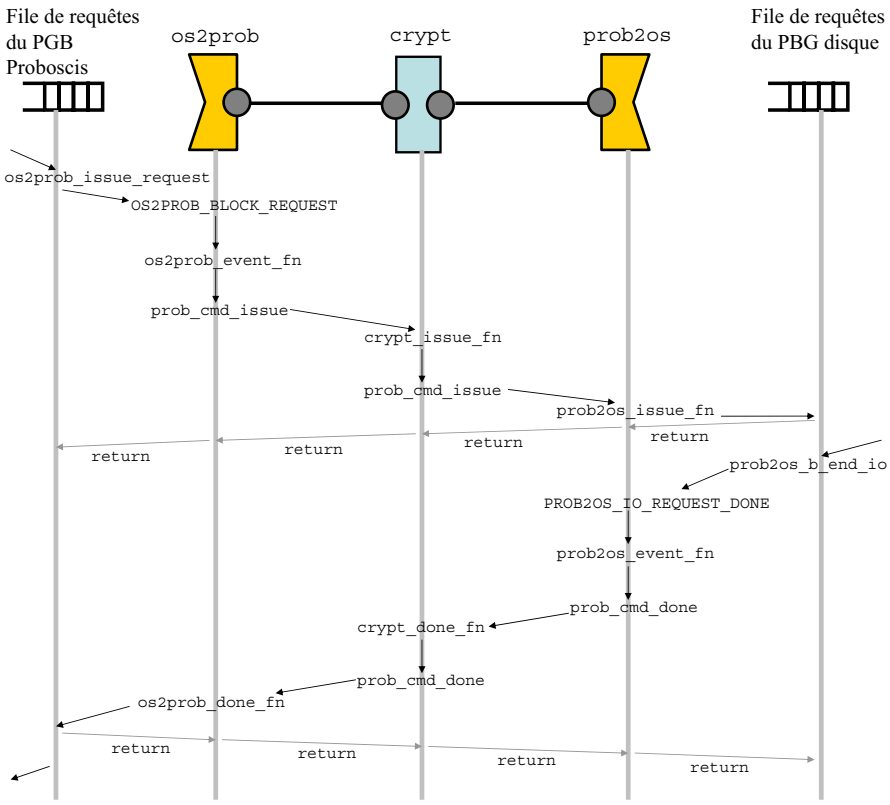


Figure 4.3 – Exemple de flot d'exécution dans un contexte centralisé

d'exécution peut alors consulter la file des tâches en attente et en extraire un nouvel événement à traiter.

Lorsque la requête d'E/S émise vers le disque est terminée, le système d'exploitation appelle `prob2os_b_end_io`, le traitant enregistré par l'instance `prob2os`. Ce traitant détermine à quelle paire instance-commande est associée la requête et demande l'émission d'un événement (de type `PROB2OS_IO_REQUEST_DONE`) sur cette instance, en passant en paramètre l'identifiant de la commande concernée. A un moment donné, le thread d'exécution Proboscis va extraire cet événement de la file des tâches et appeler le traitant d'événement de l'instance `prob2os`. Ce traitant acquitte la commande en la renvoyant en direction racine, par un appel à `prob_cmd_done`, ce qui entraîne un appel à la fonction `done_fn` de l'instance de cryptage. Celle-ci décrypte les données lues (dans le cas d'un requête de lecture) et remonte la commande vers l'instance de niveau supérieur, en appelant à son tour `prob_cmd_done` sur l'unique port de sa face racine. Ceci déclenche l'appel de la méthode `done_fn` de l'instance `os2prob`, qui acquitte la requête du système d'exploitation initialement émise sur le PGB associé au chemin et détruit la commande interne à Proboscis. Ceci achève le flot de traitement amorcé lors de la terminaison de la requête d'E/S sur le disque.

4.3.3.3 Mécanismes complémentaires pour les configurations réparties et les tâches annexes

La section précédente a détaillé le flot d'exécution au sein d'un chemin dans le cadre d'une configuration centralisée. Nous décrivons ici des caractéristiques complémentaires de l'infrastructure

d'exécution qui permettent de répondre aux besoins particuliers de certains composants, notamment au sujet des configurations réparties, de certaine tâches annexes et de la gestion du découplage entre le contexte d'exécution du programme appelant et celui du système de stockage.

Flots d'exécution complémentaires Certaines extensions ont des besoins qui ne peuvent pas être satisfaits par le modèle d'exécution que nous venons d'expliciter. Ces problèmes sont liés au fait qu'une extension est programmée comme une machine à états non bloquante. Or, si nous refusons (pour des raisons d'efficacité) les solutions par attente active, certaines actions sont intrinsèquement bloquantes ; nous en donnons deux exemples ci-dessous.

- Une extension réseau basée sur le protocole TCP/IP doit attendre, de manière bloquante, l'arrivée de commandes émises depuis un nœud distant, ou tout simplement des demandes de connexion[14].
- Une extension qui fournit une fonction de cache de blocs, basée sur le *buffer cache* du nœud qui l'héberge doit prendre en compte certains problèmes potentiels de synchronisation. Si un autre contexte d'exécution est déjà en train d'accéder à l'un (ou plusieurs) des tampons d'E/S visés au sein du cache, ceux-ci sont verrouillés et il est nécessaire de se bloquer pour être prévenu de leur déverrouillage[15].

Pour répondre à ce type de problèmes, notre modèle d'exécution autorise un composant à définir un ou plusieurs threads « assistants », qui peuvent être employés pour effectuer des actions bloquantes à la place du thread d'exécution `probed`. Reprenons les deux exemples précédents pour illustrer ce principe.

- Une extension réseau basée sur TCP/IP utilise un thread pour attendre les demandes de connexions en provenance d'autres nœuds. Lorsqu'une connexion est établie, la *socket* correspondante est transférée à un second thread assistant, chargé d'attendre l'arrivée de commandes. Quand une commande arrive, un événement est transmis à l'instance réseau. C'est alors le thread `probed` qui prend le relais en extrayant l'événement de la file des tâches en attente et en exécutant le traitant d'événement approprié, ce qui a pour effet de démarrer une chaîne d'appels qui propage la commande le long (de la portion locale) du chemin.
- Dans le cas où un tampon du cache de blocs est verrouillé, le thread principal (`probed`) transmet la commande concernée à un thread assistant (via une file de messages) et suspend son traitement de la commande (le thread principal est donc disponible pour traiter un nouvel événement). Le thread assistant récupère la commande à traiter et se bloque sur le tampon verrouillé. Lorsque l'accès au tampon devient possible, le système d'exploitation réveille le thread assistant, qui signale au thread principal, par le biais d'un événement, que le traitement de la commande peut être poursuivi.

Un thread assistant peut être instancié de manière unique ou non sur un nœud donné, ainsi qu'à différents moments. Nous détaillons maintenant ces divers cas de figures.

- La situation la plus simple correspond à un thread assistant créé et démarré à chaque fois qu'une instance d'extension est déployée. Un tel thread assistant est associé à une seule et unique instance et n'interagit qu'avec elle.

[14] En revanche, il n'y a généralement pas de problème de ce genre avec l'interface de programmation fournie par le pilote (ou les couches de communication de bas niveau) d'un réseau à hautes performances (tel que SCI ou Myrinet). En effet, il est possible d'associer un traitant à un type de message, ce qui permet d'utiliser la même technique que celle employée pour l'interface avec les requêtes du système d'exploitation décrite en 4.3.3.2.

[15] Là encore, le problème pourrait être contourné par une scrutation périodique (des tampons verrouillés) mais cette approche n'est pas satisfaisante du fait de son manque d'efficacité.

- Un thread assistant peut également être associé à une « classe d'extensions ». Dans cette situation, il y a un seul exemplaire du thread assistant (par type d'extension) sur chaque nœud, et ce thread interagit avec toutes les instances d'un même type d'extension déployées sur un même nœud. Un thread assistant « de classe », peut être démarré à différents moments :
  - lors de l'enregistrement du type d'extension concerné : ceci est indispensable pour certaines extensions réseau qui nécessitent un thread en attente de connexions pour amorcer le processus de déploiement (voir 4.4) ;
  - lors de la création de la « première » instance du type d'extension concerné : lorsqu'une instance est créée, elle ne créée un thread assistant (associé à son type) que s'il n'en existe pas déjà un.

Certaines classes d'extensions peuvent utiliser plusieurs threads assistants aux propriétés différentes. Reprenons l'exemple de l'extension réseau basée sur le protocole TCP/IP. Celle-ci nécessite tout d'abord un thread assistant « de classe » pour gérer les connexions nécessaires au déploiement de chemins répartis (comme expliqué plus haut). Elle a également besoin d'un second type de thread assistant pour attendre les commandes envoyées vers un chemin donné (et donc les messages reçus sur une *socket* donnée). Plusieurs solutions sont possibles pour ce second thread assistant : créer un exemplaire par connexion ou créer un seul thread pour toutes les connexions, ou encore utiliser le même thread pour gérer les demandes de connexion et les connexions établies (via l'usage d'une primitive `select` ou `poll`). Notre modèle de programmation autorise ces diverses solutions, qui ont chacune leurs forces et faiblesses (plus simple à programmer ou plus efficace). Il est cependant recommandé aux programmeurs de restreindre au maximum le nombre de threads assistants (et donc d'éviter d'associer un thread à chaque instance) afin de limiter la consommation de ressources induite par le système de stockage ainsi que les changements de contexte qui pénalisent les performances.

Pour finir sur ce point, précisons que l'infrastructure Proboscis emploie elle même des threads assistants chargés de certaines tâches périodiques (en particulier, le service de nommage et le service d'administration décrits plus loin). Contrairement aux threads assistants utilisés par les extensions, ceux-ci ne sont pas utilisés pour effectuer des actions bloquantes mais plutôt pour dissocier clairement les différentes activités de l'infrastructure.

Tâches non liées à une instance Ils est parfois nécessaire d'exécuter une tâche qui ne peut être associée au contexte particulier d'une instance ou d'un thread assistant (par exemple lors de la destruction d'un chemin, de l'établissement d'une connexion ou encore du déblocage de commandes en attente). À cet effet, l'infrastructure fournit la primitive `prob_sched_work`, qui permet de demander l'exécution d'une telle tâche. Cette fonction ajoute une requête d'exécution dans la file d'attente du thread principal (`probed`). En conséquence, une tâche non liée à une instance doit fournir la même garantie qu'un traitant de commande ou d'événement : être non bloquante.

Changement de contexte entre le système d'exploitation et l'infrastructure Dans l'exemple décrit en 4.3.3.2, nous avons étudié un chemin limité (en entrée comme en sortie) au niveau d'interface « blocs ». Dans une telle configuration, il n'y pas de problème pour gérer le changement de contexte d'exécution entre le système d'exploitation et l'infrastructure de Proboscis : à chaque extrémité du chemin, il y a une file d'attente (d'un côté, la file de requêtes du PGB associé au chemin, à l'autre bout, celui du disque), qui sert de point d'interface entre le(s) flot(s) d'exécution du système d'exploitation et celui de Proboscis. Selon le niveau d'interface considéré, ce problème n'est cependant pas toujours résolu de manière implicite. C'est notamment le cas pour une interface de niveau « fichiers ». Prenons l'exemple d'un processus utilisateur qui effectue une opération de lecture sur un fichier. L'opération de lecture se traduit par un appel système qui traverse d'abord la couche VFS

(cf. 2.1.3) puis les méthodes propres au SGF déployé sur le point de montage cible de la requête de lecture. Tous les traitements qui viennent d'être mentionnés sont effectués sous la forme d'une chaîne d'appels synchrones, dans le contexte du processus appelant. Contrairement au niveau « blocs », il n'y a donc pas de découplage entre le contexte de l'appelant et le contexte dans lequel sont exécutés les traitants du SGF.

Ainsi, il est dans ce cas nécessaire d'employer un « talon » (enregistré en tant que SGF auprès du système d'exploitation) pour effectuer un découplage entre le contexte d'exécution de l'appelant et celui de l'infrastructure de stockage. Ce talon a pour mission principale[16] de créer un objet « requête » correspondant à l'opération demandée et à insérer cette requête dans une file d'attente (d'où elle sera extraite par le thread `probed` pour être envoyée à l'instance racine du chemin concerné) et éventuellement de bloquer le contexte d'exécution appelant (dans le cas d'une opération d'E/S synchrone).

Dans le cas d'un chemin dont l'interface de sortie (côté « source ») est de niveau fichier (cas d'un système clients-serveur à la NFS), il n'est pas nécessaire d'introduire un point de découplage dans le sens inverse au niveau de l'instance terminale. Celle-ci doit cependant utiliser un ensemble de threads assistants pour effectuer les opérations d'E/S sur le SGF local qui peuvent être bloquantes.

Ces deux points sont illustrés sur les figures 4.4 et 4.5.

4.4 Processus de déploiement d'un chemin

L'infrastructure qui sert de base à Proboscis se présente sous la forme d'un module à charger au sein du noyau de tous les nœuds d'une grappe. Cette section commence par décrire la procédure d'enregistrement d'un nouveau type d'extension au sein d'un nœud. Elle précise ensuite les étapes de la création et de la destruction d'un chemin, ainsi que le fonctionnement du service de nommage qui simplifie le processus de déploiement.

4.4.1 Enregistrement de types d'extensions et de commandes

Une précondition nécessaire au déploiement d'un chemin est que les différents types d'extensions à instancier doivent être « connus » par les nœuds concernés. Ceci englobe deux aspects complémentaires :

- la distribution des modules (associés aux types d'extensions concernés) sur les nœuds,
- sur chaque nœud, l'enregistrement des modules auprès de l'infrastructure Proboscis.

Le premier aspect (ainsi que la définition précise de ce que nous entendons par « module ») sera abordé en 6.2. Nous nous intéressons ici à l'enregistrement d'un nouveau type d'extension sur un nœud.

Le programmeur d'une extension doit implémenter deux méthodes, `ext_init()` et `ext_cleanup()`, appelées respectivement lors de l'enregistrement et du désenregistrement de l'extension.

La méthode `ext_init()` a pour rôle d'effectuer les initialisations globales liées au type de l'extension considérée. Il s'agit notamment :

- de réserver des ressources qui serviront à l'ensemble des instances du type considéré. Pour certains besoins particuliers, les ressources peuvent être allouées immédiatement (c'est-à-dire sans attendre la création de la première instance du type d'extension). C'est, par exemple, le cas du thread en attente de connexions utilisé par l'extension réseau TCP/IP (cf. 4.3.3.3). Autre

[16] Ce talon est également responsable de l'amorce du processus de déploiement d'un chemin, lorsqu'un SGF est monté.

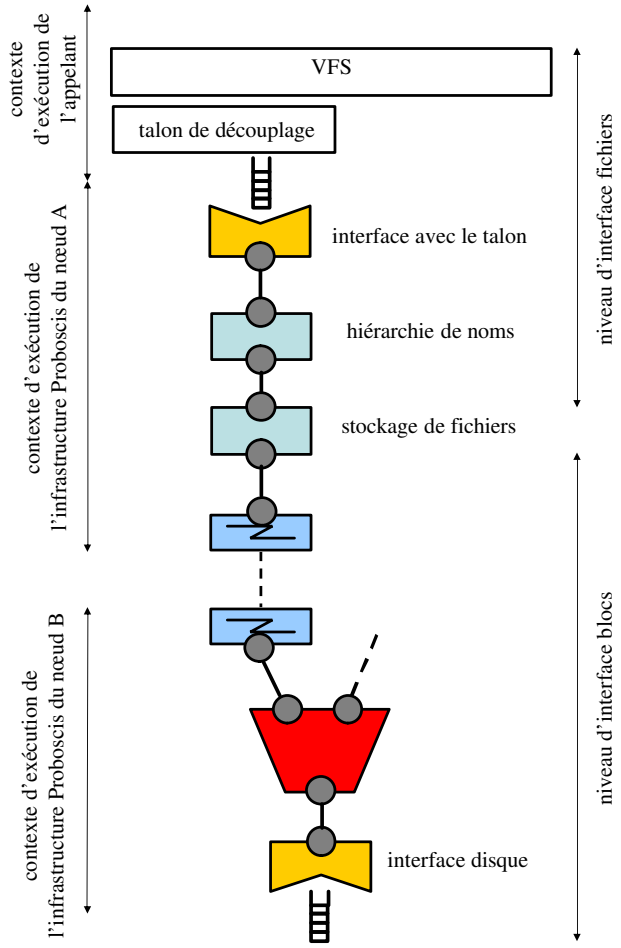


Figure 4.4 – Découplage entre le contexte d'execution appelant et celui de l'infrastructure Proboscis

exemple : la méthode `ext_init()` d'une extension racine qui exporte une interface PGB va réserver une plage d'identificateurs de périphérique, qui seront attribués aux instances lors de leur création. Dans la plupart des cas, il y a seulement une pré-réservation des ressources : des objets « vides » sont créés et leur contenu n'est véritablement alloué que lorsqu'une première instance est créée et tente d'y accéder (principe similaire à celui du « défaut de page » d'une mémoire virtuelle). Ainsi ces ressources, telles qu'un thread assistant, ou un allocateur mémoire optimisé (*slab allocator* [29]), ne sont allouées que lorsqu'elles s'avèrent vraiment nécessaires.

- d'enregistrer les types de commandes nécessaires au bon fonctionnement de l'extension (voir 4.3.1.1 pour les détails de ce mécanisme).

La méthode `ext_cleanup()` permet de libérer les ressources globales allouées à la classe d'extension et d'effacer les nouveaux types de commandes que cette dernière avait enregistrés.

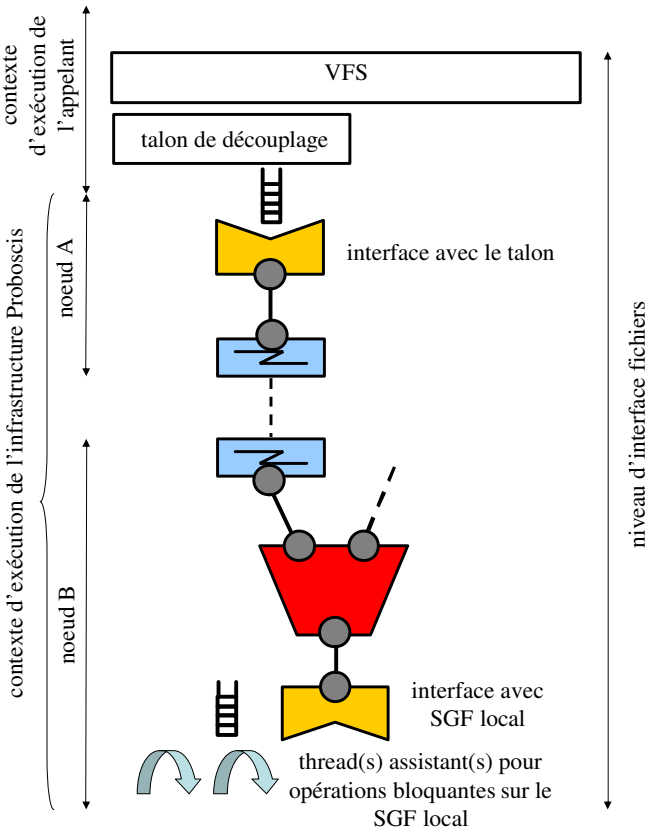


Figure 4.5 – Découplage entre les différents contextes d'execution d'un serveur de fichiers construit avec Proboscis

4.4.2 Création et destruction d'un chemin

Sur chaque nœud, l'infrastructure fournit une interface de contrôle qui permet de créer, détruire ou reconfigurer un chemin[17]. Nous décrivons ici les étapes du processus de création et de destruction d'un chemin.

4.4.2.1 Création

Une demande de création doit être émise au sein du nœud sur lequel doit être déployée l'instance racine du chemin. Elle est accompagnée d'une description textuelle qui indique la composition du chemin souhaité ainsi que la valeur des attributs publics de chaque instance[18]. Lorsque l'infrastructure reçoit une demande de création, elle convertit la description textuelle en un graphe d'objets (nommé *squelette*) représentatif de la structure du chemin à construire. Chaque objet est associé à une instance à créer et stocke tous les paramètres qui ont été spécifiés.

[17] Dans le cas de notre prototype, l'interface de contrôle est implémentée sous la forme d'appels système `ioctl` envoyés au module noyau de l'infrastructure Proboscis.

[18] La description est de la forme `/nom_ext_1 nom_param_1=valeur_1 nom_param_2=valeur_2/nom_ext_2...` Un opérateur particulier permet de déclarer les différents sous-chemins qui sortent d'une extension de dispersion : `.../nom_extension_dispersion/[sous_chemin_1 :sous_chemin_2 :sous_chemin_3]`.

L'infrastructure commence par créer la première instance (racine) du chemin puis crée une commande de type `CREATE` et la transmet à cette instance. En réaction, celle-ci exécute sa méthode `issue_fn` qui appelle le traitant approprié. Ce traitant effectue les initialisations nécessaires (en particulier, il crée un « conteneur d'état » associé à l'instance) puis propage la commande sur son port source. L'appel à `prob_cmd_issue` est intercepté par l'infrastructure, qui crée l'instance suivante et lui transmet la commande (en ayant au préalable mis à jour l'indicateur qui spécifie l'état courant du déploiement par rapport au squelette). Le processus est réitéré pour toutes les instances du chemin. Dans le cas d'une extension de dispersion, la commande de création est diffusée sur tous les ports source de l'instance (le nombre de ports source à créer sur l'instance est déduit à partir du squelette).

Chaque type d'extension réseau doit fournir les fonctions nécessaires au déploiement d'une configuration répartie. Lorsque le processus de création arrive à un stade où une extension réseau doit être créée, la partie locale de l'instance est allouée et la commande de création lui est envoyée. Celle-ci doit alors établir une connexion bidirectionnelle avec le nœud concerné, ce qui aboutit à la création de la seconde moitié (côté source) de l'instance réseau sur le second nœud. La commande de création est finalement transmise (via le réseau) à cette deuxième partie de l'instance, qui peut la propager sur son port source.

La première phase de la création s'achève lorsque la ou les instances terminales ont été créées. La commande de création est alors acquittée et parcourt le chemin en sens contraire (en direction de la racine)[19]. Chaque instance traversée par la commande acquittée exécute sa propre méthode `done_fn`, qui permet d'achever l'initialisation de l'instance. Ceci est, en particulier, utile pour récupérer des informations sur la partie aval du chemin remontée par la commande de création. Par exemple, une extension terminale servant d'interface avec un disque peut adjoindre à la commande `CREATE` des informations (taille et nombre de blocs du périphérique) qui pourront être consultées (et éventuellement modifiées ou complétées) par toutes les instances traversées par la commande acquittée. Chaque appel à `prob_cmd_done` est intercepté par l'infrastructure, qui établit alors une liaison (cf. 4.2.6) entre l'instance émettrice et l'instance destinataire de la commande. Le fait de lier les instances lors du retour de la commande de création permet d'assurer l'intégrité structurelle du chemin (les composants ne sont liés que si l'ensemble des ressources qui leur sont nécessaires ont pu être allouées).

Le trajet retour d'une commande de création permet aussi de mettre à jour les méta-données du squelette de chaque instance. En particulier, chaque instance peut ainsi connaître l'identifiant de toutes les instances situées en aval d'elle (dans la direction source) ; l'instance racine récupère ainsi les identifiants de toutes les instances du chemin.

Quand la commande atteint son point de départ, le chemin est considéré comme déployé. Un identifiant globalement unique lui est alors associé et il devient possible d'y émettre d'autres commandes, qu'il s'agisse de commandes d'administration (générées via l'interface de contrôle ou « spontanément » par l'une des instances) ou des commandes d'accès au données, créées par l'instance racine suite à une requête émanant d'un autre contexte d'exécution.

4.4.2.2 Destruction

Le processus de destruction d'un chemin est relativement semblable à celui utilisé pour sa création. L'émission d'une commande de destruction empêche l'émission ultérieure de toute autre commande (aucune autre commande ne peut être émise à partir de la racine et les commandes générées en un autre endroit du chemin seront annulées à la rencontre d'un tronçon en cours de destruction) et force chaque instance à « relâcher » toutes les commandes qu'elle avait mises en attente (ceci permet

[19] Il peut y avoir plusieurs commandes si le chemin inclut au moins une extension de dispersion. Chaque instance de dispersion attend le retour de toutes les commandes « filles » qu'elle a émise avant d'acquitter la commande de création qu'elle a initialement reçue.

par exemple de forcer la synchronisation d'un cache avec le support de stockage secondaire).

La libération des ressources allouées aux instances, la destruction de leurs informations d'état et la destruction des liaisons sont effectuées sur le « trajet retour » de la commande. Ainsi, le chemin n'est véritablement détruit que si toutes les instances sont dans un état compatible avec l'ordre de destruction (ce qui est vérifié à l'aller de la commande). Dans le cas contraire, la destruction échoue.

4.4.3 Service de noms

Un système de nommage géré par l'infrastructure permet d'associer un nom à la description d'un chemin. Ceci offre plusieurs avantages :

- utilisation de noms symboliques courts et compréhensibles (par opposition à la description complète de la structure d'un chemin),
- spécification implicite du placement de certaines parties d'un chemin,
- déploiement simplifié pour les architectures partagées (c'est-à-dire pour lesquelles plusieurs clients accèdent aux même supports de stockage via une extension d'agrégation).

Cette section explique l'intérêt du service de nommage par rapport au processus de déploiement. La mise en œuvre de ce service est décrite en 6.1.

Lorsqu'une association entre un nom et la structure d'un chemin est ajoutée à la base de données du service de nommage, il est nécessaire de préciser si le chemin concerné doit être considéré comme *libre* ou *fixe*. Dans le premier cas, le chemin est déployé à partir du nœud d'où émane la demande de création (et peut exister en de multiples exemplaires). Dans le second cas, le chemin est déployé sur le nœud fixe auquel il est associé (et ne peut exister qu'en un seul exemplaire).

Illustrons maintenant le principe et l'intérêt du service de nommage par un exemple. Un administrateur souhaite déployer un système de stockage qui permet à plusieurs clients (nœuds A et B) d'accéder à un PGB exporté par un serveur (nœud C)[20].

L'administrateur commence par définir deux chemins pour l'aider dans sa tâche :

- le chemin fixe (sur le nœud C) nommé `shared_disk` associé à la description structurelle `/share policy=roundrobin/prob2os major=3 minor=1`,
- le chemin libre nommé `client` associé à la structure `/os2prob minor=1/{shared_disk}`[21].

L'administrateur lance ensuite sur le nœud A une demande de création du chemin `/{client}`. Puisque ce nom n'est pas associé à un chemin fixe, la création du chemin débute sur le nœud courant (A). Le système de déploiement commence par contacter le service de nommage pour obtenir la traduction du nom en une description de la composition du chemin. Puisque la description retournée (`/os2prob minor=1/{shared_disk}`) commence par un nom d'extension existant, le déploiement peut commencer : l'instance `os2prob` est créée et la commande de création lui est transmise. Lorsque l'infrastructure intercepte l'appel de cette instance à `prob_cmd_issue` (pour propager la commande), elle remarque que la suite de la description correspond à un nom symbolique et contacte à nouveau le service de nommage pour obtenir une traduction. Le service de nommage constate que le nom `shared_disk` est associé à un chemin fixé sur un nœud (C) différent de l'appelant (A). En conséquence, il répond à la requête en préfixant le nom symbolique reçu avec une extension réseau : `network root_addr=<addresse de A> sink_addr=<addresse de`

[20] Par souci de simplicité, cet exemple ne présente que deux clients et un serveur mais il peut être étendu sans peine à toute configuration n-m (clients-serveurs)

[21] Au sein d'une description textuelle, un nom est distingué d'un type d'extension par l'utilisation d'accolades.

`B>/{shared_disk}`[22]. Le processus de création se poursuit et atteint le nœud C. Après le déploiement de l'instance réseau, le service de nommage est à nouveau contacté et traduit `shared_disk` en `/share policy=roundrobin/prob2os major=3 minor=1`, puis le déploiement se termine comme expliqué en 4.4.2.

Pour le second client (nœud B), le déploiement commence de la même manière[23]. Lorsque la commande atteint le nœud serveur (après la création de la partie source de l'instance réseau), le chemin *fixe* `/share policy=roundrobin/prob2os major=3 minor=1` n'est pas créé à nouveau car il a déjà été déployé. La commande de création est simplement transmise à l'instance `share`, qui alloue un nouveau port et acquitte la commande.

La notion de chemin fixe peut être complétée par des informations de contrôle, et ainsi permettre d'effectuer un filtrage sur les chemins qu'un client est autorisé à déployer sur un nœud serveur. Reprenons l'exemple précédent, en supposant que le nom `shared_disk` ait été déclaré accessible pour le client A et interdit pour le client B auprès du service de nommage. Si le client B essaye de créer le chemin correspondant au nom `client`, le processus de déploiement échouera lors de l'étape de traduction du nom `shared_disk` sur le nœud serveur (car la demande de traduction émane d'une commande originaire du nœud B, qui n'est pas autorisé à accéder à ce chemin fixe).

Ce mécanisme de filtrage permet de restreindre les possibilités de création de chemins, afin de lutter contre d'éventuelles erreurs de l'administrateur ou de tentatives d'accès non souhaitées aux données stockées sur un nœud. Il peut être complété par d'autres mécanismes plus fins, par exemple l'emploi d'une extension de contrôle d'accès qui analyse les commandes d'accès aux données pour vérifier si un client dispose des droits nécessaires pour lire ou écrire les données concernées (par exemple un fichier, un objet, ou une plage de blocs, selon le niveau d'interface considéré).

4.5 Mécanismes avancés de routage réseau

La section 4.2 a décrit comment notre modèle de composition représente de manière explicite la liaison entre deux nœuds distincts. Ceci permet de configurer finement et distinctement les propriétés de chaque canal de communication. Nous explicitons ici quelques avantages complémentaires qui découlent de cette approche (et rejoignent certains aspects des piles de protocoles configurables construites à partir du x-kernel).

- Un client connecté à plusieurs serveurs peut utiliser différents protocoles (et/ou différentes stratégies de transfert de données) pour communiquer avec chacun d'eux. Inversement, un serveur peut supporter différents protocoles de communication et être ainsi accessible à une large gamme de clients (voir figure 4.6). Même s'il est souvent souhaitable de gérer des configurations homogènes (plus simples à administrer et optimiser), de nombreuses grappes sont composées de machines hétérogènes en raisons d'acquisitions incrémentales. Le fait de représenter chaque liaison distante offre les moyens à un administrateur de tirer le meilleur parti d'un assemblage de nœuds dont les capacités différent significativement[24].
- Deux nœuds équipés de multiples cartes réseau (de technologie identique ou non) peuvent utiliser plusieurs canaux parallèles pour accélérer le transfert d'importantes quantités de données

[22] Le type d'extension réseau à utiliser est choisi automatiquement en fonction des différentes interfaces de communication disponibles sur A et C (et d'une liste hiérarchique de préférences fournie par l'administrateur.)

[23] L'administrateur ne peut utiliser le nom `client`, que si la structure du chemin à déployer sur B est strictement identique à celle utilisée pour le nœud A (en particulier au niveau des paramètres). Dans le cas contraire, il est nécessaire de définir un autre nom (associé à une description qui peut englober `{shared_disk}`)

[24] Cette remarque ne s'applique d'ailleurs pas uniquement aux capacités de communication des machines. Il est par exemple possible de spécifier des politiques de répartition de charge qui sollicitent moins certains serveurs dont les disques ou les processeurs sont plus lents.

(les canaux peuvent être configurés de manière homogène ou non). Il suffit pour cela d'utiliser un couple d'extensions représentées sur la figure 4.7. Au niveau du client, l'extension de dispersion `NW_LB` implémente une politique de distribution de charge sur les différents canaux ; au niveau du serveur, l'extension `NW_gather` sert simplement à agréger le trafic issu des différents canaux. La seule contrainte imposée par le modèle de communication est qu'une commande doit emprunter le même canal pour ses trajets aller et retour.

- Même pour une technologie matérielle donnée, un nœud peut utiliser différents protocoles de communication en fonction de ses interlocuteurs. La figure 4.8 représente un serveur équipé d'une carte réseau Ethernet et relié à deux clients. Le premier client est « directement » relié au serveur (via un commutateur) et utilise un protocole basé sur Ethernet (à la HyperSCSI, cf. 2.2) pour communiquer avec le serveur. Le second client est séparé du serveur par une infrastructure réseau plus complexe et utilise un protocole basé sur IP.

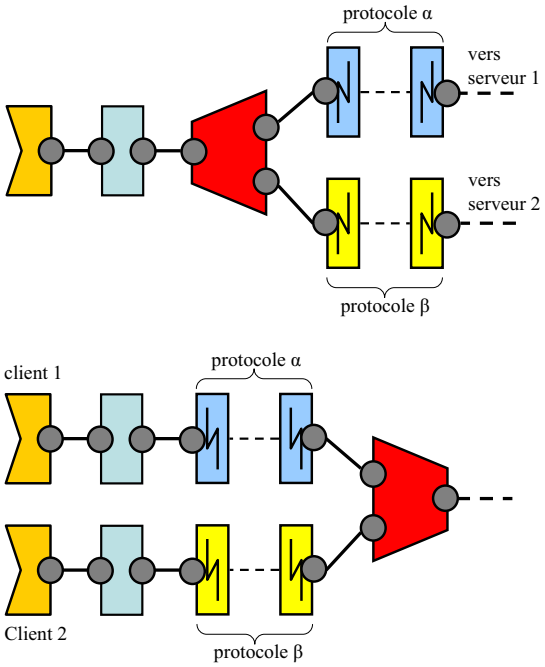


Figure 4.6 – Utilisation de canaux de communication hétérogènes par un client ou un serveur

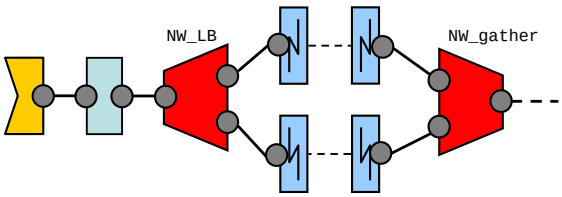


Figure 4.7 – Utilisation de plusieurs canaux de communication parallèles entre un client et un serveur

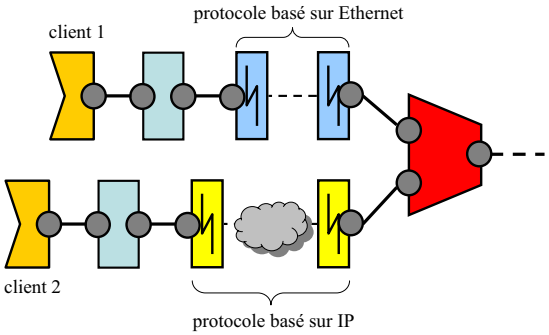


Figure 4.8 – Utilisation de différents protocoles pour une même carte réseau

4.6 Discussion

Cette section a pour objectif de situer notre proposition par rapport à d'autres travaux antérieurs, et de discuter nos choix de conception.

Nos propositions ont été influencées par les résultats de plusieurs projets de recherche. Parmi ceux-ci figurent, bien sûr, les systèmes de stockage modulaires présentés au chapitre précédent mais également d'autres canevas développés pour des contextes voisins ou plus généraux. Nous y ferons référence de manière plus précise au fil de cette section. Nous nous contentons pour le moment de les énumérer (par ordre chronologique) en précisant leur domaine d'application :

- **x-kernel :** piles de protocoles réseau optimisées [103, 161],
- **Scout :** systèmes d'exploitation pour équipements multimédia embarqués/intégrés [154, 153],
- **Click :** routeurs adaptables [66],
- **Think :** systèmes d'exploitation pour équipements contraints [68].

Cette discussion est organisée selon trois axes : le modèle de composition, le modèle d'exécution, la sécurité et la sûreté de fonctionnement.

4.6.1 Modèle de composition

4.6.1.1 Chemins

Notre notion de chemin est fortement inspirée par celle développée dans le cadre du projet Scout. Nous partageons notamment la volonté de modéliser explicitement les différents flux de données qui traversent un système afin de mieux identifier et contrôler les ressources qu'ils utilisent. Cependant, notre approche présente plusieurs différences significatives par rapport à Scout, explicitées ci-dessous. Nous en profitons également pour préciser nos divergences avec les autres canevas présentés dans l'état de l'art.

Objectifs Dans Scout, une motivation majeure de la notion de chemin est qu'elle offre une vue globale du contexte de déploiement, qui permet d'optimiser la chaîne de traitement de données (malgré sa structure modulaire). Ainsi, il est possible de remplacer une séquence de composants par un seul composant optimisé qui fournit l'ensemble des fonctionnalités attendues[25]. Nous nous intéressons également à l'apport de la vue globale que procure un chemin mais,

[25]Dans Scout, les optimisations (ou plutôt transformations) sont effectuées de façon statique, lors de l'assemblage de composants.

dans notre cas, en raison des capacités réflexives qui peuvent en découler, pour faciliter la reconfiguration dynamique et l'auto-administration d'un service. Il y a également une différence au niveau de la durée de vie des chemins. Dans Scout, un chemin est associé à une séquence d'événements relativement brève ; par exemple, un chemin est créé à l'ouverture d'un fichier ou d'une connexion et détruit lors de la fermeture correspondante. Dans Proboscis, un chemin est associé à l'utilisation d'un périphérique de stockage ou au montage d'un SGF ; il est ainsi déployé pour une période bien supérieure aux chemins de Scout.

Structure Dans le cas de Scout, une configuration bâtie à partir d'une interconnexion d'instances de modules peut être complexe et éventuellement cyclique. Dans notre contexte plus spécialisé, un chemin ne peut avoir qu'une structure arborescente.

Répartition Dans Scout, un chemin est restreint à un contexte centralisé : il ne peut s'étendre sur plusieurs nœuds. Pour offrir une vue complète d'une configuration répartie, nous pensons qu'un chemin doit représenter un service de stockage de bout en bout. Contrairement à Swarm et Abacus, il nous paraît également important de modéliser explicitement les communications sur le réseau afin d'offrir une grande souplesse de configuration et d'optimisation.

Reconfiguration Les chemins de Scout ne sont pas reconfigurables dynamiquement (il est uniquement possible de créer et de détruire des chemins lors de l'exécution). Il en va de même pour les configurations crées avec les systèmes de fichiers empilables, Swarm et x-kernel. Comme nous l'avons vu, les modifications dynamiques permises par Abacus sont limitées à la migration de tâches entre client et serveurs. Enfin, Click permet de remplacer une configuration par une autre de manière atomique en transférant l'état des anciennes instances vers les nouvelles. Ceci se rapproche de nos préoccupations mais nous nous intéressons à des systèmes potentiellement répartis et à grande échelle. Dans un tel contexte, il est souhaitable que les opérations de reconfiguration ne soient appliquées qu'aux parties du système qui en ont besoin et que les portions d'un chemin qui ne nécessitent aucune modification n'aient pas à être réinstanciées.

4.6.1.2 Granularité d'un composant

Comme nous l'avons vu en 4.2.2, notre modèle de composition a été conçu dans l'idée d'utiliser des composants de granularité moyenne. Dans une certaine mesure, un développeur de composants peut, s'il le souhaite, ignorer cette recommandation et travailler à un autre niveau de granularité.

- Il n'y a pas de restrictions au niveau de Proboscis pour développer des composants englobant d'avantage de fonctionnalités. Une telle démarche risque cependant de restreindre la finesse de paramétrage et de reconfiguration des systèmes de stockage ainsi obtenus.
- En revanche, l'architecture des chemins imposée par notre modèle (structure arborescente, sans cycle), complique et limite la tâche d'un programmeur désireux de travailler à un grain plus fin. Ceci ne constitue pas, à nos yeux, une lacune importante car un système de stockage réparti présente généralement une sémantique assez simple, et il ne nous semble pas justifié de bâtir une configuration donnée à partir de plusieurs dizaines de types distincts de composants, contrairement à d'autres contextes applicatifs (les routeurs construits avec Click, par exemple). Notre argumentation à ce sujet sera poursuivie au fil des sections suivantes.

4.6.2 Modèle d'exécution

4.6.2.1 Modèle de programmation

Le modèle de programmation utilisé pour les composants Proboscis est différent de celui adopté par les systèmes de stockage modulaires présentés au chapitre précédent (systèmes de fichiers empi-

lables, Swarm, Abacus). Ceux-ci sont basés sur un modèle procédural : un composant transmet une requête au composant suivant en appelant l'une de ses méthodes. Une requête traversant une pile de protocoles s'apparente ainsi à une chaîne d'appels de méthodes. Une méthode fournie par un composant peut être bloquante car deux requêtes concurrentes sont associées à deux contextes d'exécution distincts (on parle aussi de modèle *multithreads*[26]).

Par opposition, dans Proboscis, les composants communiquent par échange de messages. En conséquence, un composant doit être programmé à la manière d'une machine à états (finis), dont les transitions sont associées à l'arrivée d'un message ou à l'occurrence d'un événement interne au composant. Pour être conforme au formalisme d'automate, le traitant exécuté lors d'une transition d'état doit être non bloquant. En contrepartie, un seul contexte d'exécution est suffisant pour traiter des requêtes concurrentes.

L'opposition entre la programmation procédurale et la programmation par événements a, depuis longtemps, donné lieu à de nombreux débats au sein de la communauté scientifique. Lors de la dernière décennie, ces discussions ont notamment proliféré autour de sujets tels que l'architecture de noyaux de systèmes d'exploitation [63, 73] ainsi que la conception de serveurs internet (serveurs web, serveurs de courrier électronique, messagerie instantanée, etc.) de très grande puissance [165, 232, 228, 58]. La programmation par événements est généralement considérée plus efficace mais plus complexe à appréhender. Récemment, des chercheurs de l'Université de Berkeley ont « viré de bord » et préconisé l'emploi du modèle *multithreads* en vantant sa simplicité et en montrant qu'une implémentation optimisée d'une bibliothèque de threads peut fournir de bonnes performances [229, 230]. Cet argumentation rejoint le point de vue d'une (assez ancienne) publication de Lauer et Needham [132], qui souligne le caractère dual de ces deux modèles et affirme qu'un programme peut être implémenté aussi efficacement avec l'un ou l'autre. Les auteurs concluent ainsi que le choix du modèle de programmation ne doit pas être basé sur des considérations liées à l'efficacité, mais sur des critères annexes.

Nous ne remettons pas en cause ces affirmations générales sur la programmation multithreads, tant au niveau de sa simplicité que de sa potentielle efficacité. Nous avons cependant opté pour un modèle de programmation par événements pour les composants Proboscis. Ce choix est motivé par les arguments ci-dessous.

- Notre motivation principale est liée à la mise en œuvre des opérations de reconfiguration dynamique. Sans entrer dans les détails qui font l'objet du chapitre 6, il est important de comprendre que la reconfiguration d'un service internet est plus aisée que celle du service de stockage sous-jacent[27]. Un serveur peut être temporairement arrêté et reconfiguré sans grande difficulté car pendant cette période d'indisponibilité, les requêtes peuvent être redirigées sur l'un de ses pairs, dans l'hypothèse ou ce dernier a accès au même ensemble de données (et à d'éventuelles informations d'état associées aux sessions clientes en cours). Ceci est possible si les serveurs sont reliés à un système de stockage partagé. Nous nous intéressons justement à la reconfiguration du système de stockage partagé, qui s'avère plus délicate car la haute disponibilité de ce service est la pierre angulaire sur laquelle reposent les couches applicatives supérieures.

  Un formalisme d'automate à états nous semble mieux adapté qu'un modèle multithreads pour la spécification et la mise en œuvre de protocoles de reconfiguration. Ce n'est d'ailleurs pas un hasard si de nombreux services pour lesquels la haute disponibilité constitue un aspect critique (opérateurs de télécommunications, systèmes embarqués, etc.) sont basés sur un modèle de

[26] Lorsqu'aucune précision n'est donnée, nous utilisons l'expression « multithreads » pour désigner (par abus de langage) de multiples processus légers (*threads*) ou de multiples processus (lourds).

[27] Pour être plus précis, les deux extrémités de la chaîne de traitement caractéristique d'un service internet sont difficiles à reconfigurer : la gestion des connexions clientes et le service de stockage partagé. Nous ne nous intéressons qu'à ce second aspect.

programmation par événements.

- La notion de *chemin* que nous avons définie correspond à un assemblage de routeurs au sein duquel transitent des requêtes. Ce concept se traduit naturellement en termes de messages envoyés d'un routeur à un autre, et donc de programmation par événements. De plus, le modèle de programmation par événements introduit un découplage entre les différentes phases d'un protocole, ce qui simplifie la composition et l'interposition de services et permet ainsi de construire plus facilement des systèmes adaptables.
- Notre canevas vise la construction de systèmes de stockage variés. Certains systèmes sont déployés sur des nœuds entièrement consacrés au service de requêtes d'E/S mais d'autres configurations peuvent reposer sur des nœuds serveurs qui n'offrent qu'une partie de leur ressources au système de stockage. Dans ce cas, il est important d'identifier et de contrôler finement les ressources allouées au serveur de stockage. A cet effet, nous pensons qu'un modèle de programmation par événements, qui limite notamment le nombre de contextes d'exécution et manipule explicitement les informations d'état associées à chaque objet (instance de composant, requête), s'avère le plus approprié.
- Accessoirement, nous pensons (ou plutôt spéculons) que la programmation et le déboguage de composants écrits sous forme de machines à états peuvent être simplifiés par l'emploi de techniques de génération automatique de code et d'outils de vérification. Cet argument est développé dans le chapitre de conclusion.

Pour finir, rappelons que notre modèle autorise un composant à utiliser un ou plusieurs threads assistants pour effectuer des opérations bloquantes. En ce sens, notre modèle de programmation se rapproche de l'architecture AMPED (*Asymmetric Multi-Process Event-Driven*) sur laquelle sont basés certains serveurs web [165].

4.6.2.2 Ressources d'exécution

Comme nous l'avons vu en 4.3.3, l'infrastructure d'exécution de Proboscis repose sur un thread unique, et fonctionne à cet égard de manière similaire à l'infrastructure de Click (version monoprocesseur). Nous avons préféré ce principe simple à d'autres solutions (un ou plusieurs threads associés à chaque instance de composant) pour les différentes raisons exposées ci-dessous.

- L'utilisation de plusieurs threads pour exécuter les traitants d'une instance donnée complique la programmation des composants et/ou de l'infrastructure d'exécution en raisons des contraintes de synchronisation qu'elle introduit.
- Nous nous intéressons principalement à des plates-formes matérielles ne consacrant qu'une partie limitée de leur ressources au système de stockage (cas d'un système à image unique, par exemple). Ainsi, nous considérons des situations où un seul support d'exécution physique (processeur) est consacré au système de stockage. Dans ces conditions, il n'est pas intéressant d'utiliser un thread par instance déployée (ni un ensemble de n threads avec $1 < n <$ nombre d'instances sur le nœud) car une telle architecture induit d'avantage de changements de contexte sans offrir de gains d'exécution (en l'absence de parallélisme matériel).

  De plus, même en mettant de côté le coût des changements de contexte, une architecture basée sur un thread par instance force à utiliser systématiquement les files de messages pour la communication entre instances, ce qui augmente la latence de la traversée d'une commande. En utilisant un seul thread, il est possible de « court-circuiter » les files de messages pour propager une commande lorsque le chemin est « libre » (c'est-à-dire lorsqu'il n'est pas nécessaire de mettre la requête en attente au cours de sa traversée du chemin).

En contrepartie, le fait de reposer sur un thread unique restreint la fréquence des opportunités de prise de décision d'ordonnancement[28], mais ceci n'apparaît pas comme un inconvénient majeur. En effet, les chemins sont généralement de taille réduite (5 à 10 instances) et les traitements effectués par chaque composant assez brefs, si bien que l'ordonnanceur de l'infrastructure d'exécution est appelé à intervalles raisonnablement proches.

- Comme expliqué précédemment, l'utilisation d'un seul thread simplifie également le contrôle des ressources consommées par le système de stockage. L'activité d'éventuels threads assistants est directement reliée à celle du thread d'exécution[29], qui constitue donc un levier central (manipulable à la fois par l'ordonnanceur du système d'exploitation et par lui-même).
- Enfin, en complément du modèle de programmation par événement, le recours à un thread unique permet de simplifier les mécanismes de reconfiguration d'un chemin.

Notre choix d'utiliser un thread unique pour traverser une succession de modules rejoint les modèles employés par x-kernel et Scout. Dans le cas du premier, c'est le même contexte d'utilisation (originaire d'un appel système par un programme utilisateur) qui traverse toutes les couches d'une pile de protocole. Scout associe également un seul thread au parcours d'un chemin, mais chaque chemin est associé à un flux distinct. Dans le cas de Proboscis, un chemin peut agréger les flux de requêtes issus de différents clients et a une durée de vie plus longue.

Nos propositions relatives au modèle d'exécution ne sont aucunement définitives. Il reste plusieurs pistes de recherche importantes à ce sujet, notamment au niveau du contrôle de flux global à un chemin et à la gestion optimisée d'architectures multiprocesseur dédiées à la tâche de serveur de données. Ces enjeux sont explicités dans le chapitre de conclusion.

4.6.3 Sécurité et sûreté de fonctionnement

Cette section aborde les préoccupations majeures de tout administrateur d'une grappe. Nous commençons par décrire les mécanismes prévus à ce sujet dans notre proposition. Dans un second temps, nous explicitons les lacunes courantes de notre approche ainsi que des pistes permettant d'y remédier.

La section 4.4.3 a montré comment le système de nommage permet de contrôler quels chemins peuvent être déployés au sein de la grappe, et de manière complémentaire, comment une extension de contrôle d'accès peut intercepter des tentatives de lecture ou d'écriture de données interdites. Une extension de contrôle d'accès peut également être utilisée pour protéger les données stockées sur disque contre les tentatives erronées d'écriture d'un client défaillant (panne byzantine). Ainsi, il n'est plus nécessaire de disposer d'un équipement hautement spécialisé (commutateur SAN et/ou contrôleur de disque administables) pour effectuer un filtrage de requêtes (opération de *fabric fencing*, cf. 2.3.3), préférable à l'arrêt brutal du nœud défaillant (qui nécessite d'ailleurs aussi du matériel spécifique).

Notre infrastructure présente cependant plusieurs lacunes dont nous tentons d'évaluer la gravité ci-dessous.

Tout d'abord, il n'existe pas de mécanisme robuste permettant de sécuriser le déploiement de l'infrastructure Proboscis sur l'ensemble des nœuds d'une grappe. Nous faisons l'hypothèse que les binaires associés au noyau de l'infrastructure Proboscis peuvent être distribués et installés de manière

[28] Cette restriction s'applique uniquement aux décisions d'ordonnancement internes à l'infrastructure Proboscis, dont les ressources d'exécution sont à tout moment susceptibles de réquisition par le système d'exploitation.

[29] Une exception notable concerne les threads assistants employés par certaines extension réseau (par exemple TCP/IP), qui sont chargés de recevoir les commandes envoyées par d'autres nœuds. Il n'est pas possible de réguler l'activité de ces threads en agissant uniquement sur le thread d'exécution de Proboscis. Ceci requiert une gestion globale (répartie) du contrôle de flux au sein d'un chemin. Cette problématique est évoquée dans les perspectives du chapitre de conclusion.

sécurisée et fiable sur tous les nœuds concernés. Ce postulat ne semble pas déraisonnable car la plupart des grappes sont utilisées dans un environnement réseau relativement isolé du monde extérieur avec des politiques d'accès clairement établies (en particulier au niveau des droits d'administration).

Par ailleurs, notre modèle ignore également les problèmes de falsification d'identité (par exemple, un client usurpant l'identité d'un autre pour contacter un serveur ou le système de nommage et interceptant les réponses avant qu'elles ne parviennent au nœud « imité »). Cet aspect ne nous paraît pas crucial dans la majorité des cas (grappe isolée de l'extérieur et utilisée pour un tâche à la fois) mais peut s'avérer sensible dans le contexte d'un centre d'hébergement (*hosting center*), où une grappe peut être partitionnée entre plusieurs applications. Ce problème, qui concerne surtout les protocoles de communications basés sur l'adressage IP[30], peut généralement être traité au niveau du filtrage réseau (réseau privé virtuel, par exemple) mis en œuvre par le système de virtualisation de grappe (cf. 1.2.2.3). Si nécessaire, la sécurité pourrait encore être renforcée par l'intégration de mécanismes d'authentification au sein de Proboscis (sans compliquer à outrance la tâche de l'administrateur car chaque nœud d'une grappe dispose déjà d'une paire de clés de cryptographie utilisées par des protocoles comme SSH).

Enfin, notre infrastructure et les différentes extensions qui peuvent y être insérées se présentent sous la forme de modules à charger par le noyau du système d'exploitation. A ce titre, il existe un risque de compromettre l'intégrité du système (et des informations stockées) par le biais d'un module malveillant. Le milieu des années 1990 a été le théâtre d'une forte activité de recherche autour de la construction de systèmes extensibles sûrs. Certains systèmes (comme SPIN [25]) s'appuient sur des extensions écrites dans un langage sûr, d'autres (tels Vino [200]) inspectent et modifient les binaires avant de les charger et encapsulent tous les appels à une extension au sein d'une transaction.

Pour notre part, nous avons choisi d'ignorer le problème spécifique des extensions volontairement malveillantes (boucles infinies, traitants bloquants, altération du système, etc.), qui nous semble plus sensible dans le cadre des ordinateurs personnels que dans celui des serveurs de données[31]. Il nous paraît cependant important d'envisager une réponse aux fautes qui peuvent découler d'un bogue (involontaire) au sein d'une extension du noyau. Plutôt que de contraindre les programmeurs d'extensions à utiliser un langage spécialisé ou de dégrader significativement les performances du système en utilisant des transactions, nous sommes plutôt attirés par des approches récentes qui permettent d'isoler les fautes d'un pilote de périphérique (par rapport au reste du noyau ainsi qu'aux autres extensions) et de redémarrer le pilote fautif. C'est notamment le cas des projets Nooks [217, 216] et Xen [74] (qui permet en outre de multiplexer un pilote vers différentes instances de systèmes d'exploitation encapsulées dans des machines virtuelles), capables de détecter et survivre à la plupart des fautes logicielles, en ne nécessitant pas (ou très peu) de modifications du code des pilotes, et en causant un surcoût tout à fait acceptable. Ce type de mécanisme pourrait protéger les nœuds d'un serveur en grappe contre la majorité des pannes causées par Proboscis et être utilisé pour isoler des chemins distincts traversant le même nœud. Il resterait toutefois à développer, au dessus de ce substrat de base, un mécanisme de tolérance aux fautes logicielles permettant de reconstruire les chemins affectés et d'assurer la cohérence des méta-données qui y sont associées.

[30] Les protocoles utilisés pour les réseaux optimisés sont moins sujets à ce genre d'attaques car une grande partie du routage des messages est effectuée par le matériel.

[31] Ceci peut sembler paradoxal. Nous entendons par là qu'un « utilisateur lambda » est d'avantage exposé à ce risque car il est plus à même (étant généralement peu informé et peu vigilant) de récupérer et d'installer des modules malveillants. Par opposition, un administrateur est beaucoup plus précautionneux à cet égard et n'installe que des pilotes (et autres extensions) dont il considère l'origine comme fiable. Par ailleurs, de nombreux serveurs sont désormais basés sur des systèmes d'exploitation *open-source*, qui permettent un audit complet du code pour se prémunir de ce type de danger.

4.7 Bilan

Ce chapitre a défini un canevas extensible basé sur des composants logiciels, pour simplifier la construction de services de stockage répartis, sous la forme de *chemins* (pipeline arborescent de composants) que traversent les requêtes d'entrées/sorties échangées entre une entité cliente et un organe de stockage. Le concept de chemin semble bien adapté pour modéliser les fonctions caractéristiques de la chaîne de traitement qui relie client et serveur(s) : traduction d'une requête d'un niveau d'interface à un autre, agrégation/éclatement/ordonnancement de requêtes, gestion de caches, transport des requêtes et des données sur le réseau.

Dans le triple objectif de maximiser les performances, maîtriser la consommation de ressources et simplifier la reconfiguration d'un chemin, nous avons opté pour un modèle de programmation par événements pour les composants logiciels employés. Chacun d'entre eux doit donc être implémenté sous la forme d'une machine à états dont les transitions sont non bloquantes (des threads « assistants » peuvent être employés pour effectuer les tâches bloquantes).

Le modèle d'exécution choisi associe un thread unique à l'ensemble des instances de composants déployées sur un nœud. Cette approche permet de simplifier l'ordonnancement des tâches ainsi que le contrôler des ressources associées à l'infrastructure de stockage.

Le système de déploiement d'un chemin repose sur un service de nommage qui permet de résoudre progressivement la correspondance entre un nom symbolique et la composition de la structure à créer, ainsi que de prendre automatiquement en charge les contraintes de répartition et de partage de composants. Le travail des administrateurs humains est ainsi facilité.

Chapitre 5

Gestion configurable et optimisée des transferts intensifs de données au sein d'une grappe

Sommaire

5.1 Introduction

Un service[1] axé sur la manipulation de données réparties se doit d'être efficace. Pour atteindre cet objectif, il est nécessaire d'optimiser les transferts de données en fonction des capacités du réseau d'interconnexion sous-jacent. Ceci aboutit généralement à une implémentation du service très dépendante de l'interface de programmation fournie par la technologie d'interconnexion considérée et d'une stratégie de transfert de données précise. En conséquence, la maintenance du code associé au service devient fastidieuse dès qu'il s'agit de gérer plusieurs types d'interfaces réseau et/ou plusieurs schémas de communication.

[1] Au cours de ce chapitre, les termes « service » et « application » sont employés au sens le plus large possible. Il peut s'agir d'un programme s'exécutant en mode utilisateur (application « finale », intergiciel) ou dans le contexte du noyau. Notre infrastructure prototype est implémentée au niveau du noyau.

Ce chapitre propose une infrastructure permettant de découpler le code applicatif (et en particulier les messages de contrôle) de la spécification des aspects liés aux transferts de données . Cette approche offre plusieurs avantages, énoncés ci-dessous.

- **Les communications liées au flux de contrôle (requêtes/réponses, messages de synchronisation, etc.) et celles associées aux transferts de données proprement dits peuvent être optimisées de façon distincte**. Ceci s'avère important car ces deux types de communications ont des caractéristiques nettement différentes en termes de taille et de gestion de ressources. Les messages de contrôle sont généralement de petite taille (quelques kilo-octets au plus) et optimisés pour une faible latence. Par opposition, un transfert de données peut englober plusieurs méga-octets d'information et être optimisé pour différents critères : latence, consommation des ressources (CPU, mémoire, bande passante) au niveau de l'émetteur et/ou du récepteur, etc.
- Si l'infrastructure physique de la grappe permet une communication directe entre tous les nœuds, il est possible de « court-circuiter » la topologie logique de l'application et de transférer directement des données entre une source et un puits. Ainsi, **les nœuds intermédiaires (qui n'ont qu'un rôle de routage de messages) ne sont plus impliqués lors des échanges de données entre un client et un serveur**. Ceci permet d'éviter certaines recopies des données pour, au final, améliorer la latence globale des transferts et décharger les nœuds routeurs.
- **Un nœud destinataire de données peut retarder l'amorçage d'un transfert afin d'optimiser la gestion de ses ressources locales**. Ainsi, un serveur peut supporter un pic de charge de façon plus aisée, grâce à un meilleur ordonnancement de ses activités et une consommation mémoire mieux contrôlée.
- Il devient possible de **modifier (de manière statique ou dynamique) le type d'interface ainsi que la stratégie de communication employés** pour le transfert des données entre plusieurs nœuds, sans impact sur le code du service qui effectue les transferts de données.

L'infrastructure proposée permet à un nœud (« primaire ») d'exporter des tampons de mémoire vers d'autres nœuds (« secondaires »). Pour cela, le noeud primaire commence par coupler les tampons dans un espace d'adressage global à la grappe puis envoie un descripteur de tampon (relatif à l'espace global) au nœud secondaire concerné. Celui-ci peut alors coupler les tampons dans son espace d'adressage local afin de lire ou modifier leur contenu. Lorsque le couplage local est supprimé, les éventuelles mises à jour des données sont (si nécessaire) propagées sur le nœud primaire.

La flexibilité de l'approche tient au fait que l'interface de programmation utilisée par les applications fournit uniquement des primitives de manipulations de tampons et de gestion des couplages associés, les transferts de données nécessaires étant gérés par l'infrastructure sous-jacente. Ainsi, la mise en œuvre d'un couplage local peut être réalisée de différente manières, et optimisée pour divers types de matériel et de contraintes applicatives.

Cette proposition découle de nos travaux sur Proboscis mais son champ d'application n'est, à nos yeux, pas restreint au contexte du stockage réparti. D'autres types de services ou applications pour grappes basés sur l'usage intensif de grandes quantités de données peuvent probablement bénéficier d'une telle approche. Par conséquent, les premières sections de ce chapitre sont volontairement générales (même si certains exemples concrets ont trait à la thématique du stockage). L'intégration de cette infrastructure au sein de Proboscis est explicitée en 5.5.

La section 5.2 présente les concepts de base sur lesquels se fonde notre infrastructure. Leur utilisation est illustrée par des exemples applicatifs détaillés en 5.3. La section 5.4 décrit différentes stratégies de transfert qu'il est possible d'implémenter avec le modèle de programmation proposé. Enfin, nous comparons en 5.6 notre approche aux travaux existants.

Remarque complémentaire : les idées développées dans ce chapitre n'abordent pas les problèmes de sécurité. Nous faisons en particulier l'hypothèse que les différents nœuds d'une grappe s'accordent une confiance mutuelle. Dans le cas contraire, des mécanismes de filtrage d'accès peuvent être mis en œuvre en amont de notre infrastructure, soit dans les couches basses de la pile réseau de chaque nœud (cas des communications par paquets, par exemple TCP/IP sur Ethernet), soit au niveau d'un commutateur/routeur (cas d'un réseau spécialisé avec d'éventuelles capacités de RDMA tel qu'Infiniband).

5.2 Principes de base

Nous nous intéressons ici à la simplification et à l'optimisation des échanges de données stockées en mémoire (dans des « tampons d'E/S ») entre des composants logiciels déployés sur des nœuds distincts. Notre infrastructure est fondée sur trois mécanismes principaux : la notion d'*assemblage de tampons* (*buffer aggregates*) qui permet de simplifier la manipulation des données et d'en limiter le nombre de recopies, la définition d'*espaces d'adressage globaux* associés à un ensemble de politiques de transfert et l'emploi d'une couche de communication abstraite pour l'échange de descripteurs de données.

5.2.1 Assemblage de tampons

Une application intensive en données doit manipuler des tampons d'E/S le plus efficacement possible, en évitant au maximum, voire totalement, les recopies de données. Parmi les opérations critiques, on peut citer l'ajout d'en-têtes protocolaires, la fusion de plusieurs tampons ou, au contraire, l'éclatement d'un tampon en sous parties. Pour répondre à ce besoin, nous avons développé la notion d'*assemblage de tampons* (ou BA pour *Buffer Aggregate* par la suite) et un ensemble d'opérations associées. Cette idée est inspirée des travaux du projet IOLite menés à Rice University [166, 167] et s'en rapproche par de nombreux aspects.

Un BA regroupe un ensemble ordonné de couples <*pointeur, taille*> identifiant chacun un tampon de mémoire. Nous employons le terme « tranche » (*slice*) pour désigner chaque élément au sein d'un BA. Un BA contient également un ensemble de méta-données, notamment au sujet de l'état des tampons (vides, modifiés ou non depuis leur création) et à celui des relations de parenté entre BA (mise en jour en fonction des opération de fusion et d'éclatement)[2]. Un BA est, en outre, associé à un nœud d'origine et à un identifiant unique au niveau de la grappe. Nous présentons ci-dessous un sous-ensemble représentatif de l'interface de programmation associée aux BA.

- `iobuf_create` crée et initialise un BA. Si la taille des données spécifiée est nulle, le contenu du BA est laissé vide. Dans le cas contraire, un ensemble de tampons (vides) est alloué.
- `iobuf_import` est similaire à `iobuf_create` mais au lieu d'allouer de nouveaux tampons, elle fait référence à des tampons existants (alloués par une couche supérieure du système ou de l'application) qui lui ont été passés en paramètre.
- `iobuf_append` permet d'ajouter une tranche à un BA.
- `iobuf_remove` permet de retirer une tranche d'un BA.
- `iobuf_copy` permet de créer un nouveau BA « fils », qui référence certains tampons d'un ou plusieurs BA « parents ».

[2] En l'absence de précision, nous parlons par la suite du « contenu » d'un BA pour désigner les données qu'il englobe et non ses méta-données.

- `iobuf_destroy` détruit un BA ainsi que les tampons vers lesquels il pointe, sauf dans le cas ou d'autres BA font encore référence à ces tampons ou si les tampons ont été initialement importés (et non alloués par l'infrastructure).
- `iobuf_gen_start` retourne un objet (« contexte d'accès ») permettant d'interroger le contenu d'un BA.
- `iobuf_gen_next_read` retourne l'adresse et la taille de la tranche suivante (relative au décalage global — par rapport au début du BA — qui lui est passé en paramètre). Dans la plupart des cas, le contenu d'un BA est parcouru de manière séquentielle. Le décalage courant associé à un object contextuel peut cependant être modifié pour accéder directement à une tranche précise du BA.
- `iobuf_gen_end` détruit un contexte d'accès associé à un BA.

5.2.2 Espace d'adressage global et mécanismes de transfert

5.2.2.1 Principes généraux

La section précédente a présenté l'interface employée pour manipuler des tampons d'E/S. Nous allons maintenant décrire comment ces tampons sont rendus accessibles pour l'ensemble des nœuds d'une grappe. Notre approche est basée sur la définition d'un espace d'adressage global associé aux BA. Un nœud peut accéder au contenu d'un BA dont il n'est pas à l'origine en demandant la création d'un exemplaire local. Toutes les modifications effectuées sur l'exemplaire local sont, au final, répercutées sur le nœud de départ.

La mise en œuvre de cette idée implique l'introduction d'une distinction au niveau des assemblages de tampons : un BA peut encapsuler une *instance locale*, associée à des adresses de tampons en mémoire locale et/ou une *instance globale*, associée à des adresses globales à la grappe (obtenues, par exemple, en combinant un identifiant unique de nœud à une adresse locale). À un instant donné, un BA peut englober (1) une instance locale uniquement, (2) une instance globale uniquement ou (3) les deux types d'instances, en fonction de la situation considérée.

Lorsqu'un nœud X crée un BA, son contenu est uniquement accessible de manière locale. Le nœud X peut ultérieurement décider de partager ce BA avec d'autres nœuds de la grappe. Pour cela, il est, au préalable, nécessaire d'exporter le BA en le couplant à l'espace d'adressage global. Le BA est ainsi associé à la fois à une instance locale et à une instance globale. Le nœud X peut alors envoyer une référence du BA à un autre nœud (Y). Cette référence correspond en fait (au niveau du mécanisme de sérialisation fourni par l'infrastructure) à une copie de l'instance globale du BA original[3]. Lorsque le destinataire décide d'accéder au contenu du BA, un couplage est établi entre l'espace d'adressage global et celui local au nœud (la mise en œuvre du couplage est détaillée en 5.2.2.2). Ce couplage est annulé quand le nœud Y n'en a plus la nécessité et les éventuelles modifications apportées au contenu du BA sont alors répercutées sur l'exemplaire original du BA sur le nœud X.

L'établissement et la suppression d'un couplage sont gérés de façon transparente par l'infrastructure de partage. L'accès aux tranches de données englobées au sein d'un BA est conditionné à l'utilisation d'un contexte d'accès, comme expliqué en 5.2.1. Ainsi, les appels aux méthodes `iobuf_gen_start` et `iobuf_gen_end` permettent de délimiter la période au cours de laquelle un service accède aux tampons référencés par un BA. Plus précisément, les méthodes `iobuf_gen_start` et `iobuf_gen_end` acceptent un paramètre qui permet de spécifier si l'instance

[3] Autrement dit, le destinataire reçoit une copie du BA qui n'inclut qu'une instance globale. L'instance locale n'est pas transmise par l'émetteur car elle n'a de sens que dans le contexte local de ce dernier.

considérée est locale ou globale. Si l'instance indiquée à `iobuf_gen_start` n'existe pas, elle est automatiquement allouée. De façon similaire, lors d'un appel à `iobuf_gen_end`, s'il n'existe plus de contexte d'accès actif vers l'instance concernée, celle-ci est détruite.

Le programmeur d'un service est tenu de signaler à l'infrastructure des informations d'état sur les données englobées par le BA (à la fois lors de la création du BA et pour toute modification de cet état). Ceci permet à l'infrastructure de n'effectuer des transferts de données que lorsque cela est réellement nécessaire. Ainsi, il n'y a pas besoin de recopier le contenu de tampons initialement vides (lors de la création d'une instance locale sur un nœud destinataire) ni de mettre à jour les tampons originaux si le destinataire ne les a pas modifiés. Dans notre prototype actuel, il existe un statut unique associé à l'ensemble des tampons d'un BA : le contenu d'un BA est considéré comme intégralement vide, intact ou modifié. Ceci est suffisant dans le contexte d'un système de stockage mais d'autres applications basées sur des techniques de partage plus complexes peuvent avoir des besoins plus évolués (par exemple, modification uniquement de quelques portions spécifiques d'un BA). Notre modèle peut être facilement étendu pour ce cas de figure : il suffit d'utiliser des informations d'état d'une granularité plus fine.

5.2.2.2 Mise en œuvre pour une plate-forme particulière

Cette section explicite comment l'infrastructure de partage permet d'abstraire les opérations de couplage et d'en fournir diverses implémentations optimisées pour une plate-forme cible (interface de communication, charge) précise.

Les opérations de couplage et le format utilisé pour les adresses globales sont encapsulés au sein d'un module de code nommé *IODSM Address Space* (IAS). Un IAS est lié à (1) une interface de programmation des communications (API de communication) ainsi qu'à (2) un ensemble de méthodes de couplage implémentant une stratégie de transfert particulière. L'API de communication peut être plus ou moins spécialisée pour une catégorie de matériel (par exemple Myrinet-GM [156] ou VIA [64]) voire neutre (par exemple, basée sur des sockets TCP). Il est possible de définir plusieurs stratégies de transfert pour une même API de communication. Dans ce cas, chaque stratégie donnera lieu à une définition d'IAS distincte. L'infrastructure fournit les fonctionnalités nécessaires pour l'enregistrement et la liaison avec un IAS précis.

Le modèle de programmation associé à notre infrastructure définit un ensemble de méthodes abstraites que doit implémenter chaque IAS.

- `map_local_to_iodsm` couple un BA dans l'espace global en créant une instance globale du BA.
- `map_iodsm_to_local` rend les tampons d'une instance globale accessibles dans l'espace d'adressage local de l'appelant en créant une instance locale du BA.
- `update_iodsm_from_local` copie le contenu (modifié) des tampons locaux vers les tampons distants référencés par l'instance globale du BA.
- `update_local_from_iodsm` copie le contenu des tampons distants référencés par l'instance globale du BA vers les tampons locaux.
- `unmap_local` détruit le couplage local d'un BA.
- `unmap_global` détruit le couplage global d'un BA.

Le rôle de ces primitives pour permettre le partage de données entre nœuds est illustré sur la figure 5.1.

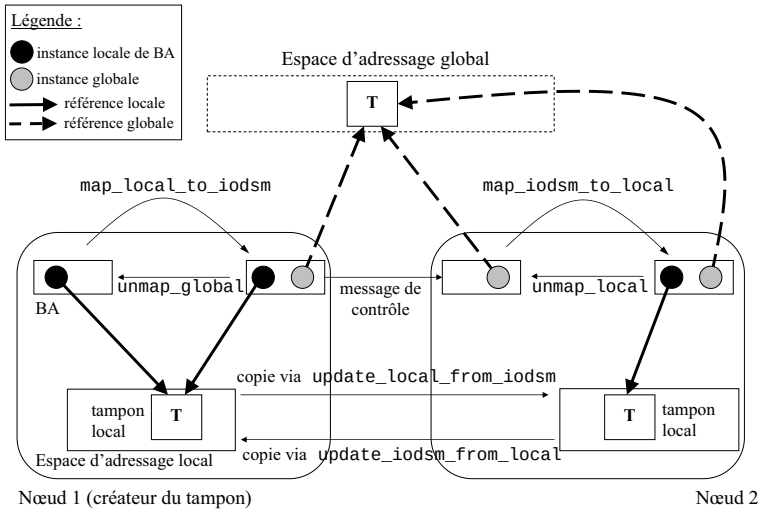


Figure 5.1 – Illustration du rôle des principales primitives d'un IAS pour le partage de données entre deux nœuds

Jusqu'à présent, nous avons volontairement laissé la notion de « couplage » relativement floue. Ceci s'explique par le fait que l'opération concrète associée à l'établissement d'un couplage local peut varier en fonction des capacités du matériel d'interconnexion choisi ainsi que de la stratégie de transfert considérée. Il peut, d'une part, s'agir d'une recopie des données du nœud distant, opérée soit par un transfert RDMA (accès distant à la mémoire), soit par envoi de message(s). D'autre part, un couplage local peut également être réalisé par un couplage de mémoire distante (qui permet d'accéder de façon transparente à la mémoire d'un autre nœud) lorsque le matériel sous-jacent le permet (comme dans le cas de SCI).

Enfin, il convient de préciser que certaines méthodes associées à un IAS sont potentiellement bloquantes, puisqu'elles peuvent induire des communications à distance. Il s'agit de `map_iodsm_to_local`, `update_iodsm_from_local` et `update_local_from_iodsm`. Etant donné le modèle de programmation adopté pour l'infrastructure Proboscis (machine à état), nous avons choisi d'exposer cette caractéristique dans l'interface de programmation d'un IAS (et, par ricochet, dans l'interface de gestion des BA). Ainsi, les méthodes susmentionnées acceptent en paramètre une fonction de *callback*, appelée lorsque l'opération est terminée. Il est ainsi possible d'obtenir un bon recouvrement entre les communications et les traitements effectués par le thread d'exécution de l'infrastructure Proboscis.

Il est néanmoins tout à fait envisageable de développer une interface synchrone pour la gestion des tampons. Ceci permettrait de simplifier la tâche des programmeurs d'applications multithreads.

5.2.3 Vue d'ensemble de l'architecture

Nous récapitulons ici l'interaction des deux briques de base décrites précédemment (assemblages de tampons et espaces d'adressage globaux), et nous précisons ensuite comment les nœuds peuvent s'échanger des références de tampons. Enfin, nous précisons les implications de ces aspects sur les différents acteurs associés à une application répartie.

La couche de gestion des assemblages de tampons interagit avec la couche de gestion des transferts via l'interface définie en 5.2.2.2. Ainsi différents modules d'IAS peuvent être utilisés sans qu'il soit besoin de modifier le code de manipulation des BA et, *a fortiori*, le code de l'application.

Ceci est possible car la manipulation des BA masque entièrement les transferts de données entre nœuds au niveau de l'application. Les seules communications explicites que le code applicatif doit gérer sont relatives à la transmission d'un BA d'un nœud à un autre, ce qui permet d'établir un lien entre les messages de contrôle (correspondant au protocole de l'application) et les mécanismes de transferts configurables fournis par l'infrastructure de partage. Dans la plupart des cas, un IAS se repose sur une fonction générique fournie par l'infrastructure pour sérialiser un BA à transmettre, qui sérialise uniquement ses méta-données et l'instance globale qui lui est associée. Cependant, chaque IAS est libre d'implémenter son propre mécanisme de sérialisation d'un BA, pour par exemple supprimer le découplage entre messages de contrôle et transferts de données[4] (voir section 5.4).

Pour parvenir à une indépendance complète du code applicatif vis à vis de de l'interface de communication sous-jacente, il faut que les échanges de descripteurs de données entre nœuds soient aussi programmés de manière générique. Cet objectif peut être réalisé de deux manières, décrites ci-dessous.

- Si l'application est organisée d'une façon modulaire, alors la contrainte de généricité peut être levée. Différents modules de communication (chacun spécialisé pour une interface) peuvent être développés et masquer les détails des communications envers le reste de l'application. C'est cette technique qui est employée au sein de Proboscis.
- Une autre solution consiste à utiliser une couche de messages (fiable) indépendante du matériel employé telle que Madeleine II [20] ou Portals [34], voire l'interface standard des sockets BSD (la majorité des fabricants de technologies d'interconnexions pour grappes fournissent désormais une interface socket à hautes performances utilisable à la fois par les applications utilisateur et les services noyau [157, 199]).

En résumé, passons en revue la chaîne des responsabilités qui aboutit au déploiement d'une application basée sur l'infrastructure que nous proposons. Un développeur d'application doit structurer son code autour de l'abstraction d'assemblage de tampons et de l'interface de programmation associée. À l'exception des messages mettant en oeuvre le protocole applicatif concerné (qui permet l'échange de descripteurs de tampons), aucun transfert de données entre différents nœuds ne requiert une prise en charge explicite par le programmeur.

Un développeur réseau fournit un ou plusieurs modules IAS, chacun étant spécialisé pour une interface de communication particulière et une stratégie de transfert adaptée à un ensemble de contraintes. Il n'est pas forcément nécessaire de développer un IAS pour chaque nouvelle application. Un IAS peut également implémenter une « méta-stratégie », c'est-à-dire choisir dynamiquement (et réévaluer de manière périodique) la stratégie de transfert à utiliser en fonction des conditions opérationnelles. Les décisions dynamiques relatives à la stratégie de transfert peuvent également être prise par le module applicatif chargé des communications.

Un administrateur choisit au moment du déploiement, en fonction de la plate-forme cible et des contraintes d'exploitation, l'IAS le mieux adapté à l'application. Précisons que les résultats observés peuvent inciter l'administrateur à modifier *a posteriori* l'IAS choisi, voire à demander le développement d'une nouvelle stratégie de transfert mieux adaptée aux conditions d'exécution. Dans tous les cas, le changement d'IAS n'a aucun impact sur le code de l'application.

[4] En d'autres termes, le découplage entre messages de contrôle et transferts de données proposé par notre modèle n'est pas obligatoire.

La figure 5.2 schématise l'architecture de l'infrastructure de partage de tampons répartis.

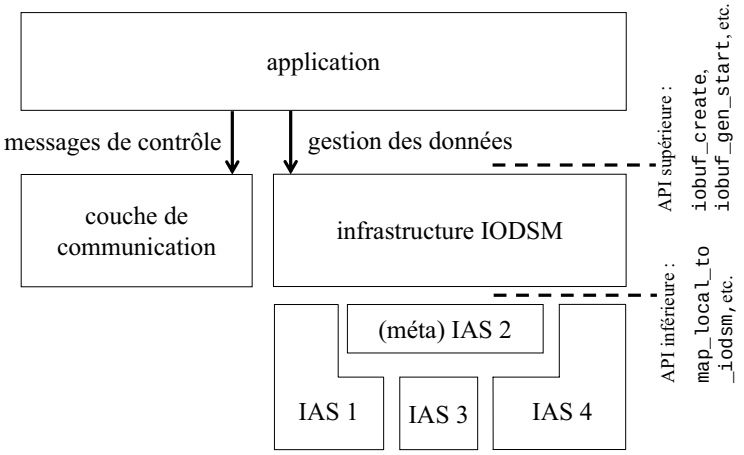


Figure 5.2 – Architecture de l'infrastructure IODSM

5.3 Exemples complets

Nous pouvons maintenant décrire en détails l'utilisation du modèle de programmation que nous proposons. Cette section présente deux exemples d'applications réalistes : un système de stockage réparti (SSR) et un service de caches coopératifs (SCC). Le SSR permet à un nœud de stocker/récupérer des données sur un ou plusieurs nœuds distants. Le SCC rend quant à lui possible la mutualisation des caches (*buffer caches*) de multiples nœuds[5]. Ces exemples sont bien sûr inspirés du domain d'application visé par Proboscis ; nous pensons cependant qu'ils sont représentatifs des besoins de la plupart des applications réparties pour grappes.

5.3.1 Service de stockage réparti

La figure 5.3 expose les différentes étapes associées à un transfert de données, que nous décrivons ci-dessous. Pour simplifier la présentation, l'exemple n'utilise que deux nœuds, un client et un serveur. Notre modèle de programmation est néanmoins adaptable sans difficultés (voire avantageux car il permet d'éviter des recopies de données inutiles sur des nœuds intermédiaires) pour des situations impliquant davantage de nœuds pour le traitement d'une requête.

1. Le prologue correspond à l'événement au niveau du client qui motive le transfert de données. Le SSR reçoit une requête (de lecture ou d'écriture) associée à un ensemble de tampons.
2. Un assemblage de tampons (BA) est créé, avec uniquement une instance locale en important les tampons alloués par la couche supérieure d'où émane la requête.
3. Le BA est couplé à l'espace d'adressage global. Le BA est désormais associé à une instance locale ainsi qu'à une instance globale.
4. Le BA est sérialisé (recopie de l'instance globale uniquement) ...
5. ... et encapsulé au sein d'une requête envoyée au serveur.

[5]Pour simplifier la présentation, seul le cas d'une requête (fructueuse) en lecture est traité. Le protocole de mise à jour des caches n'est pas abordé.

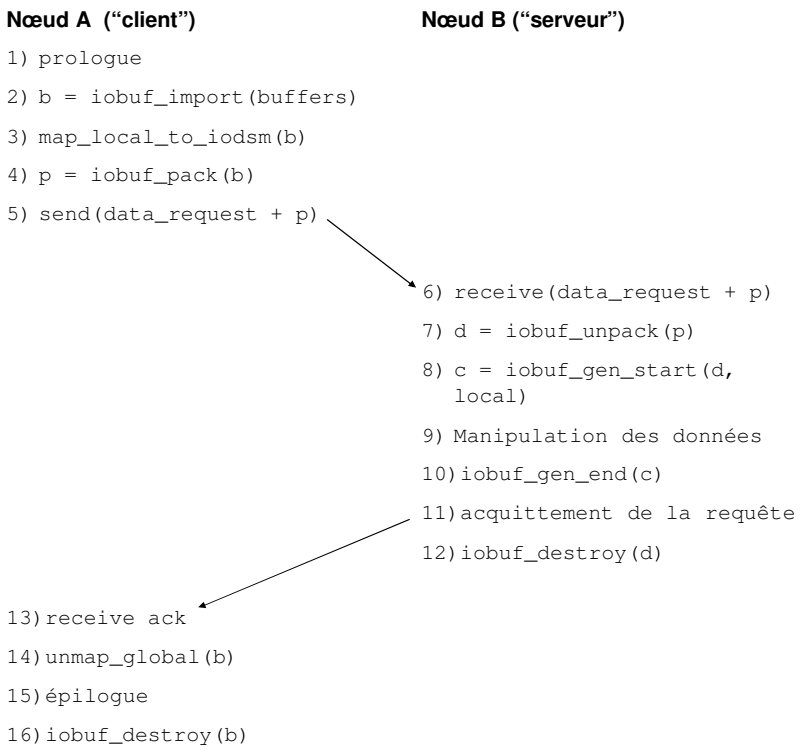


Figure 5.3 – Séquence de pseudo-code associée au flot de contrôle d'une requête

6. Le serveur reçoit la requête ...
7. ... et désérialise le BA, ce qui entraîne la création d'une instance globale sur le serveur.
8. Le serveur demande la création d'un contexte d'accès pour manipuler l'instance locale. Puisqu'il n'en existe pas encore sur le nœud B, une instance locale est automatiquement créée par l'infrastructure et un contexte est retourné au serveur. En fonction des paramètres spécifiés à `iobuf_gen_start`, l'instance locale associée au BA est automatiquement couplée sur le nœud local. Des tampons sont alloués sur le nœud serveur et, si le contenu du BA n'est pas vide (cas d'une écriture), les données présentes au niveau du client sont récupérées via un appel à `map_iodsm_to_local`.
9. Le serveur effectue les manipulations de données propres au protocole qu'il implémente en parcourant l'instance locale du BA. Dans le cas du SSR, cela consiste à récupérer les informations sur les tampons locaux (par une suite d'appels à `iobuf_gen_next_read`) et de démarrer les transferts correspondants au niveau du disque.
10. Lorsque la tâche du serveur est achevée (fin des transferts disque ou fin du couplage explicite, selon l'application considérée), il n'y a plus lieu de conserver une instance locale du BA.

    Dans le cas d'une requête de lecture, l'état des données du BA est modifié : il passe de l'état *vide* à l'état *contenu modifié*. Inversement, dans le cas d'une requête d'écriture, l'état passe de *contenu modifié* (à synchroniser sur disque) à *contenu stable*.

    Le serveur signale à l'infrastructure qu'il n'a plus besoin d'accéder à l'instance locale du BA en appelant `iobuf_gen_end`. Puisqu'il n'existe pas d'autre contexte d'accès associé à cette instance, `iobuf_gen_end` appelle automatiquement deux méthodes de l'IAS concerné.

- Tout d'abord, si le BA est dans l'état *contenu modifié*, la méthode `update_iodsm_from_local` est appelée pour propager les modifications des données au niveau des tampons du client. Sa terminaison (signalée de manière asynchrone par un mécanisme de *callback*) indique que toutes les données ont été transférées.
- Dans un second temps, l'appel à la méthode `unmap_local` supprime l'instance locale du BA.

11. Les données ayant été transmises (dans la direction appropriée selon qu'il s'agisse d'une opération de lecture ou d'écriture), la requête peut être acquittée par le serveur. Il n'est pas nécessaire de sérialiser le BA au sein du message d'acquittement.
12. Le serveur détruit alors son exemplaire du BA (y compris l'instance globale restante).
13. Le client reçoit l'acquittement ...
14. ...et supprime le couplage global du BA correspondant[6]. Il ne reste alors qu'une instance locale du BA sur le nœud A.
15. L'épilogue correspond à des traitements spécifiques à l'application (en général, la propagation de la requête acquittée en direction des couches supérieures).
16. Enfin, le BA initialement crée par le client (à l'étape n°2) peut être détruit. Les tampons associés à l'instance locale ne sont pas détruits par `iobuf_gen_end` car ils ont été importés et non créés pour l'occasion par l'infrastructure.

Remarque : La description développée ci-dessus fait l'hypothèse (aux étapes 4 à 8) que le mécanisme employé pour la sérialisation des BA correspond à l'implémentation générique fournie par l'infrastructure (seul l'instance globale du BA est sérialisée). Cependant, un IAS peut fournir sa propre implémentation de ce mécanisme, et en particulier joindre une copie des données à l'instance globale. Dans ce cas, lorsque le serveur désérialise le BA, il obtient immédiatement une instance locale et une instance globale.

5.3.2 Service de caches coopératifs

Nous allons maintenant reprendre la séquence de la figure 5.3 en précisant les différences par rapport à l'exemple précédent centré sur le service de stockage réparti (SSR).

- Étape 1 : Le SCC reçoit une requête de lecture et détermine le nœud à contacter.
- Étape 8 : Les paramètres spécifiés à `iobuf_gen_start` sont tels que l'instance locale du BA est créée mais laissée vide.
- Étape 9 : Le serveur couple, de façon explicite, les tampons du cache local à l'instance locale du BA par une suite d'appels à `iobuf_append`. Contrairement à l'exemple précédent, le couplage de l'instance locale du BA n'est pas effectué automatiquement car les données à lire sont déjà présentes en mémoire et il est souhaitable d'éviter une recopie inutile.
- Étape 10 : Pour que les données soient transférées sur le nœud d'origine du BA, le fanion d'état associé au contenu du BA est mis à jour et passe de l'état *vide* à l'état *contenu modifié*. Comme dans le cas d'une lecture avec le SSR, la méthode `iobuf_gen_end` provoque le transfert des données vers le client puis supprime l'instance locale du BA. Cependant, contrairement à l'exemple du SSR, les tampons ne sont pas détruits car ils ont été importés explicitement à partir du cache local (et non pas alloués par l'infrastructure IODSM).

[6]En fait, la requête initiale a pu être envoyée à de multiples serveurs (par exemple en raison d'un schéma de *striping* ou de duplication des données sous-jacent) via la création de BA « fils » (cf. 5.2.1). Nous supposons que l'acquittement de la requête initiale implique que toutes les requêtes filles ont été elles-mêmes acquittées au préalable.

Pour compléter l'exemple qui vient d'être développé, on peut citer la primitive `iobuf_gen_fill`. Celle-ci repose sur un appel à `update_local_from_iodsm` et permet de copier des données dans les tampons locaux d'un BA ayant été couplé de manière explicite. Elle est par exemple utilisée pour la mise à jour d'un cache (dont les tampons existent déjà).

5.4 Stratégies de transfert de données

Cette section décrit diverses réalisations d'IAS et montre ainsi comment différentes stratégies de transfert peuvent être implémentées de façon transparente au sein de l'infrastructure de communication pour supporter des types de matériels et des contraintes variés. Elle commence par présenter des stratégies optimisées pour les capacités de réseaux spécialisés à travers l'exemple de SCI. Dans un deuxième temps, elle aborde le cas de réseaux tels que la famille Ethernet, limités à une communication par échange de messages.

Avant de commencer, il convient d'expliciter la terminologie employée. Nous distinguons deux types généraux de transferts de données. En mode *push*, le transfert est démarré par l'émetteur des données. En mode *pull*, le transfert est, au contraire, à l'initiative du récepteur. Nous utilisons les termes *client* et *serveur* dans le même contexte que l'exemple détaillé de la section 5.3. Dans le cas d'un requête d'écriture, le client et le serveur correspondent respectivement à l'émetteur et au récepteur des données. Par opposition, dans le cas d'une requête de lecture, le serveur correspond à l'émetteur et le client au récepteur.

5.4.1 Stratégies de transfert pour réseaux spécialisés : l'exemple de SCI

Les cartes SCI sont, d'une part, dotées d'un moteur de RDMA bidirectionnel, qui permet à un nœud d'accéder directement à la mémoire d'un autre (en lecture ou écriture) sans accaparer leurs processeurs respectifs. Cette caractéristique est commune à la plupart des réseaux à haut débit actuellement disponibles. D'autre part, une interface SCI est également capable de coupler une portion de mémoire distante dans l'espace d'adressage local d'un nœud (ce que nous désignerons par la suite par l'expression « couplage direct »). Il est alors possible d'accéder à cette portion de mémoire distante par le biais d'entrées/sorties programmées.

Nous décrivons ci-dessous les stratégies, basées sur ces caractéristiques, que nous avons implémentées.

5.4.1.1 Stratégies basées sur des couplages directs

L'utilisation de couplages directs est particulièrement intéressante dans le cas où il n'est pas nécessaire de garder sur le nœud serveur une copie en mémoire des données transférées. Ceci correspond au cas d'un serveur qui lit ou écrit des données directement depuis un disque (sans les conserver dans un cache en mémoire[7]). Dans cette situation, il est possible de transférer directement des données entre le disque du serveur et le tampon mémoire d'un client (et vice versa) grâce à l'établissement d'un couplage direct au niveau du serveur. Dans le cas d'une requête d'écriture, le serveur établit un

[7] La décision de ne pas maintenir de cache au niveau du serveur peut être motivée par divers arguments :

- économie de ressources (mémoire, CPU) au niveau du serveur, consacré uniquement à la persistance des données ;
- le cache du client suffit à éviter la plupart des accès disques ;
- les données écrites par le client sont volumineuses mais ne sont pas lues très souvent et il n'est ainsi pas très avantageux de les conserver en cache (exemple : calcul *out-of-core* ou flux multimédia), etc.

couplage mémoire vers chaque tampon du client, puis, avec les adresses locales ainsi obtenues, démarre un transfert sur disque. Le contrôleur du disque est ainsi « trompé », car sa tentative de lecture locale des données est interceptée par la carte SCI qui effectue de façon transparente une lecture dans la mémoire du client. La même technique est valable pour une requête de lecture (dans ce cas, la carte SCI écrit les données retournées par le disque directement dans la mémoire du client). Ce principe est illustré sur la figure 5.4.

L'utilisation de couplages directs pour la mise en œuvre d'un IAS est à la fois simple en terme de code et efficace (il n'y a pas de véritable transfert ni de recopie de données). L'implémentation des méthodes IODSM est aisée, il s'agit uniquement de créer ou supprimer un couplage direct SCI et la méthode `update_iodsm_from_local` est vide. Un autre bénéfice non négligeable de cette technique est qu'elle consomme très peu de ressources au niveau du serveur : le processeur n'est pas monopolisé pour la transmission des données (le transfert a directement lieu entre l'organe de stockage et le client) et ses bus ne sont que peu sollicités (une seule traversée du bus d'E/S — pour la communication entre le disque et la carte SCI — et aucun accès au bus mémoire).

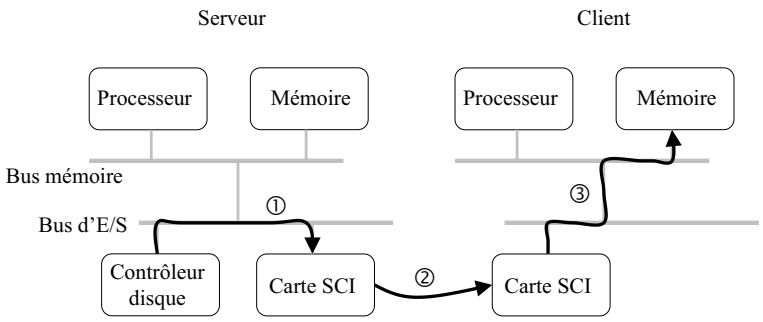


Figure 5.4 – Transfert direct de données entre un contrôleur de disque et un tampon distant via une interface réseau SCI (cas d'une lecture sur disque)

En pratique, cette approche comporte cependant son lot d'inconvénients. Tout d'abord, le nombre de couplages directs supportés simultanément par le matériel est limité si bien que l'infrastructure peut pénaliser les performances des applications qui utilisent SCI sur le même nœud[8]. En outre, si le contenu des tampons distants (couplés en mémoire locale) est directement transmis à un périphérique d'E/S, des problèmes de gestion d'erreur peuvent aboutir à l'altération des données reçues[9].

Il est également possible d'utiliser des couplages directs pour implémenter des transferts depuis ou vers la mémoire du serveur. Dans ce cas l'implémentation de l'IAS demeure assez simple (un transfert de données à distance apparaît comme une recopie en mémoire locale) mais contrairement au cas précédent, cela mobilise davantage de ressources au niveau du serveur (cycles processeur et encombrement du bus mémoire).

[8]Car le mécanisme de recouvrement, qui permet de gérer d'avantage de couplages simultanés que ce que supporte une carte SCI, est très coûteux.

[9]Ceci est lié au fait que certaines erreurs de lecture à distance (lors d'une écriture sur disque) ne peuvent être détectées qu'*a posteriori* par le serveur car le contrôleur disque n'est pas capable de les déceler et peut, par conséquent lire des données erronées.

5.4.1.2 Stratégies basées sur les capacités de RDMA

Contrairement aux couplages directs, l'utilisation du RDMA implique la recopie de données entre deux tampons distants. Ce mode de transfert présente cependant plusieurs avantages :

- Pour des quantités de données importantes, la latence de démarrage d'un transfert RDMA est compensée par le gain de temps d'un transfert groupé (alors que dans le cas des couplages directs, il faut une communication distante par mot de 32 ou 64 bits, selon le processeur employé). Avec les générations actuelles de matériel SCI, le seuil intéressant pour recourir au RDMA se situe autour de 8 ko).
- Les capacités de *scatter-gather*[10] d'une carte capable de RDMA permettent une gestion simple des assemblages de tampons (car ils reposent sur le même principe).
- La plupart des interfaces réseau à hautes performances gèrent les échanges de données par RDMA. Une stratégie de communication basée sur cette technique de transfert est donc adaptable assez facilement à différentes familles de matériel (par opposition aux couplages directs, qui ne sont actuellement permis que par la technologie SCI).

Une stratégie de transfert par RDMA peut être basée sur un mode *pull* ou *push*. Le mode pull est le mieux adapté au principe du RDMA (le récepteur décide au moment voulu de l'endroit où doivent être écrites les données attendues). En contrepartie, le récepteur paye la majorité des coûts de communication (car c'est à lui qu'il revient d'amorcer le transfert DMA), ce qui peut être parfois considéré comme un inconvénient (par exemple si le récepteur est un nœud serveur qui doit gérer plusieurs clients en parallèle.). Au contraire, avec le mode push, c'est l'émetteur qui prend en charge l'établissement du transfert mais il est dans ce cas nécessaire de maintenir un ensemble de tampons préalloués pour chaque émetteur potentiel au niveau du récepteur.

Le choix d'un mode push ou pull est surtout important pour les transferts en écriture (c'est-à-dire du client vers le serveur). Dans le cas d'une lecture, le mode push est moins contraignant (que pour une écriture) car l'émetteur (le serveur) connaît, grâce à l'instance globale du BA, l'adresse des tampons du destinataire (le client). Il n'y a donc pas besoin de maintenir un ensemble de tampons alloués en permanence au niveau du client. Néanmoins, si le coût d'amorçage du transfert par RDMA est élevé et que l'on souhaite alléger la charge du serveur, il est possible d'utiliser le mode pull dans le cas d'une lecture, en procédant de la façon suivante :

1. le serveur prépare une liste décrivant les tampons locaux où se trouvent les données à lire ;
2. cette liste est encapsulée dans un message envoyé au client ;
3. à la réception du message, le client dispose des informations nécessaires pour démarrer une lecture à distance par RDMA ;
4. lorsque le transfert est terminé, le client envoie un message d'acquittement au serveur ;
5. à la réception de l'acquittement, le serveur considère que l'opération (asynchrone) `update_iodsm_from_local` est terminée.

En pratique, cette stratégie est peu avantageuse car la latence et le coût induits par les communications supplémentaires sont généralement supérieurs aux gains escomptés.

[10] La technique de *scatter-gather* (« éclatement / dispersion [des E/S] »), permet à un périphérique de lire ou d'écrire directement une suite d'octets dans un ensemble de tampons non contigus en mémoire. La plupart des périphériques capables de DMA, localement (disque, carte Ethernet) ou à distance (carte SCI ou Myrinet), implémentent cette fonctionnalité.

5.4.1.3 Stratégie hybride

Il est également possible de définir des stratégies hybrides qui utilisent différents mécanismes de transfert en fonction de la situation considérée. On peut par exemple combiner les avantages des couplages directs et du RDMA en choisissant :

- pour les lectures : les couplages directs, afin de décharger le bus mémoire du serveur ;
- pour les écritures : les transferts par RDMA pour éviter les problèmes d'altération de données et limiter le nombre de couplages directs utilisés, en mode pull dans le but de réduire la consommation mémoire sur le serveur.

Ces choix se traduisent de la façon suivante au niveau de l'implémentation de l'IAS correspondant :

- la méthode `map_iodsm_to_local` infère si le BA considéré est utilisé pour une lecture ou une écriture (en fonction de l'état initial de son contenu — un contenu vide est associé à une opération de lecture) et utilise le mécanisme de transfert adapté ;
- de façon similaire, la méthode `update_iodsm_from_local` détermine de quelle manière a été couplée l'instance locale du BA et utilise le mécanisme de transfert approprié ;
- les autres méthodes sont basées sur le même principe et sont relativement simples à implanter.

Comme nous l'avons suggéré en 5.2.3, une étape vers davantage d'adaptabilité consisterait à implémenter un IAS capable de commuter dynamiquement entre différentes stratégies de transfert (au moins au niveau des opérations de lecture) en fonction des statistiques observées à l'exécution (nombre de couplages directs couramment utilisés, taille des tampons transférés, etc.).

5.4.2 Stratégies de transfert pour réseaux banalisés : l'exemple de TCP/IP

Toutes les grappes ne sont pas équipées de cartes réseaux à hautes performances capables de RDMA (ni de couplages directs vers de la mémoire distante). C'est en particulier le cas des générations actuelles du matériel Ethernet[11]. Dans ce cas de figure, il est donc nécessaire d'implémenter des stratégies de transfert reposant uniquement sur l'envoi de paquets.

Nous avons choisi de baser nos stratégies de transfert par envoi de paquets sur le protocole TCP/IP pour plusieurs raisons. D'une part, pour ne pas avoir à gérer explicitement au sein des IAS les aspects d'ordre, de fiabilité et de contrôle de flux. D'autre part, car cette interface est supportée par quasiment tous les types de réseaux (car les fabricants de réseaux spécialisés fournissent désormais une pile protocolaire optimisée avec une interface socket TCP/UDP). Ainsi, ces stratégies par envoi de paquets sont génériques et peuvent également être utilisées avec un réseau spécialisé[12]. Il est cependant tout à fait concevable d'implémenter des stratégies de transfert par envoi de messages optimisées à plus bas niveau pour un type de matériel précis (par exemple, les trames d'une famille Ethernet donnée).

Nous avons implémenté trois stratégies de transfert par envoi de paquets. La première est basée sur le mode push, les deux autres sont des variantes autour du mode pull.

Transfert groupé démarré par l'émetteur Les données sont jointes au message de contrôle (requête d'écriture ou acquittement d'une lecture selon le cas). Ainsi, une requête d'écriture inclut à la fois les métadonnées sérialisées d'un BA et les données contenues dans ce BA.

[11] Des travaux d'adaptation des mécanismes de RDMA pour Ethernet sont en cours depuis plusieurs années mais l'acceptation et la diffusion à grande échelle de ces améliorations ne sont pas encore d'actualité.

[12] Ceci peut être utile à un administrateur désireux d'utiliser un réseau à haut débit existant pour augmenter les performances d'une application mais ne disposant pas d'un IAS approprié, ni du temps ou des compétences pour en développer un.

Transfert progressif démarré par le récepteur Le récepteur récupère les données de manière échelonnée (une tranche après l'autre) lorsqu'il parcourt le contenu d'une instance.

Transfert groupé démarré par le récepteur Le récepteur signale à l'émetteur qu'il est prêt à recevoir les données. Ce dernier transmet alors l'intégralité des données en un seul envoi.

Ces trois stratégies ont chacune leurs forces et faiblesses. La première ne nécessite pas de synchronisation et fournit les meilleures performances en latence. En contrepartie, elle peut engendrer une importante consommation mémoire au niveau du récepteur, si celui-ci est est souvent amené à mettre en attente des requêtes. En outre, si une requête doit traverser un ou plusieurs nœuds routeurs avant d'atteindre sa destination finale, le mode push induit des recopies de données inutiles sur ce(s) nœuds intermédiaires.

Les deux autres approches corrigent les lacunes de la première en laissant le récepteur décider du moment de l'envoi des données. Cela se traduit par une latence et une consommation CPU accrues en raison des communications supplémentaires. Cet impact sur les performances des transferts est moins sensible pour la troisième stratégie, au prix d'une consommation mémoire plus importante au niveau des couches réseau.

Comme dans le cas des transferts par RDMA, le choix de la stratégie à adopter est surtout important pour les requêtes d'écriture. Dans le cas d'une lecture, le récepteur (le client) s'attend à l'arrivée des données (et sait où les placer en mémoire) et la latence prime sur la consommation mémoire. En conséquence, le mode pull fait souvent l'affaire. En outre, il est possible d'optimiser la réponse à une lecture en mode push, en évitant que le serveur attende l'acquittement du transfert par le client avant de considérer l'opération `update_iodsm_from_local` terminée (et de pouvoir ainsi acquitter la requête initiale de lecture).

- Lorsqu'il n'y a pas d'intermédiaire entre le client et le serveur, l'optimisation est triviale car le transfert de données et la requête de lecture transitent par le même canal TCP, qui garantit que les données seront reçues par le client avant l'acquittement de la requête de lecture.
- En présence d'un ou plusieurs nœuds intermédiaires, la mise en œuvre est un peu plus complexe car les canaux de communications utilisés ne sont pas les mêmes : les données sont envoyées au client par une connexion TCP directe mais la requête acquittée transite par une chaîne de connexion TCP qui traverse l'ensemble des nœuds intermédiaires. Pour garantir que l'acquittement d'une lecture n'est traité par le client qu'après la réception des données, il est nécessaire d'introduire un mécanisme de synchronisation au niveau de l'appel à la méthode `unmap_global` de l'IAS, qui doit normalement être appelée avant que le client accède aux données lues.

Gestion des connexions Puisque nous utilisons TCP/IP, les échanges de données entre nœuds sont effectués selon un mode connecté. La plupart du temps, lorsqu'une opération IODSM déclenche un transfert de données à distance, une connexion existe déjà entre les deux nœuds concernés et il n'y a donc aucune difficulté (dans le contexte de Proboscis, cela correspond à une extension réseau ayant établi un liaison explicite entre les deux nœuds). Cependant, dans certaines situations, il n'existe pas de connexion préalable entre les nœuds[13]. Dans notre implémentation, ce problème est géré à la manière d'un défaut de page : si la connexion n'existe pas, elle ext créée à la volée. Les connexions qui ont été établies de cette manière et sont restées inactives pendant un long moment sont périodiquement détectées et fermées par l'infrastructure.

[13] C'est notamment le cas pour des topologies où un routeur est intercalé entre les deux nœuds communicants (cf. 5.5) et pour un système de caches coopératifs (cf. 7.7.2), où il n'est pas nécessaire de maintenir une connexion permanente entre chaque paire de clients.

Pour finir, précisons que si l'emploi d'un mode de communication par paquets engendre forcément des recopies de données (contrairement à des transferts par couplages directs ou RDMA), l'utilisation du concept d'assemblage de tampons n'impose pas de recopies supplémentaires. En effet, l'API socket usuelle offre le concept d'`iovec` (descripteur de tampons dispersés en mémoire) qui correspond de façon naturelle à la structure d'un BA et dont la plupart des systèmes d'exploitation et pilotes de cartes réseaux (équipées d'un moteur de DMA) tirent parti pour éviter les recopies nécessaires au réassemblage des données dans une seule zone de mémoire (et vice versa). Notre implémentation exploite ce mécanisme pour envoyer les données à partir de leur tampons d'origine et aussi pour placer directement les données reçues sur un socket aux emplacements appropriés.

5.5 Application à Proboscis

Pour illustrer l'intégration de notre infrastructure de partage de données avec Proboscis, nous allons maintenant revoir le cheminement d'une requête de lecture au sein d'un chemin qui relie un client à un serveur de stockage. Nous n'allons pas décrire en détails toutes les étapes du transfert (comme en 5.3) mais uniquement les interactions entre les extensions Proboscis et l'infrastructure de partage de tampons.

La requête de lecture créée par l'extension `os2prob` (interface entre les système d'exploitation et le chemin géré par l'infrastructure Proboscis) inclut parmi ses champs une référence vers un BA. Lorsque l'extension réseau du client reçoit la requête, elle l'inspecte pour déterminer si celle-ci comporte des champs correspondant à un BA[14]. Une fois la référence au BA trouvée, elle appelle alors une méthode générique (commune à tous les types d'extensions réseau) qui, d'une part, couple le BA à l'espace d'adressage global en appelant la méthode `map_iodsm_to_local` de l'IAS concerné et, d'autre part, ajoute un marqueur dans les méta-données du BA pour identifier l'IAS auxquel il est associé. La requête est ensuite sérialisée et envoyée à l'autre moitié de l'extension réseau, déployée sur le serveur, ou elle sera désérialisée. Le reste du cheminement est facilement déductible des exemples précédents.

L'IAS utilisé par l'extension réseau est spécifié comme paramètre lors du déploiement du chemin. Il est cependant modifiable dynamiquement, soit en utilisant les techniques de reconfiguration décrites au chapitre suivant, soit en utilisant un « méta-IAS » qui commute entre différents IAS en fonction des conditions d'exécution observées.

Pour compléter ces explications, considérons maintenant l'exemple représenté sur la figure 5.5, où un routeur (nœud B) est intercalé entre le client (nœud A) et le serveur de stockage (nœud C). On peut distinguer deux cas.

- **Si les couples de nœuds A-B et B-C appartiennent au même domaine de communication** (même technologie d'interconnexion et possibilité pour chacun de communiquer directement avec l'un de ses pairs), alors les transferts de données peuvent être effectués directement entre A et C (sauf dans le cas d'écritures en mode push). L'extension réseau R2 sur B ne sert qu'à transmettre les requêtes et leurs acquittement, elle ne modifie pas les BA au sein des requêtes car ils sont déjà couplés à un espace d'adressage global valide sur le reste du chemin[15].
- **Dans le cas contraire** (si par exemple A et B sont reliés par GigabitEthernet et B et C par Infiniband), les transferts de données directs entre A et C sont impossibles. Dans ces circonstances,

[14] Ceci est possible grâce aux annotations fournies pour la définition de chaque type de commandes, cf. 4.3.1.1.

[15] En prolongeant ce raisonnement, si les périphériques de communication et de stockage étaient capables d'accéder à un espace d'adressage partagé global à la grappe, aucune recopie de données ne serait nécessaire. Ce type de matériel n'existe malheureusement pas à l'heure actuelle.

l'instance réseau R2 joue alors le rôle de passerelle entre les deux réseaux, ce qui implique une recopie des données sur le nœud B. Pour cela, chaque BA couplé à l'espace global A-B est couplé localement sur B est associé à un BA « fils », couplé dans l'espace global B-C.

Pour finir, considérons que le nœud B n'est pas qu'un simple routeur mais qu'il applique un certain schéma de répartition, par exemple du striping. Supposons également que A, B et C appartiennent au même domaine de communication. Dans ce cas, pour chaque requête du client, l'extension de dispersion sur B crée plusieurs requêtes filles (une par nœud de stockage concerné) et chacune de ces requêtes est associé à un nouveau BA, instancié à partir du BA de la requête initiale. Ainsi, tous les transferts de données peuvent être effectués directement entre le client et les nœuds de stockage[16].

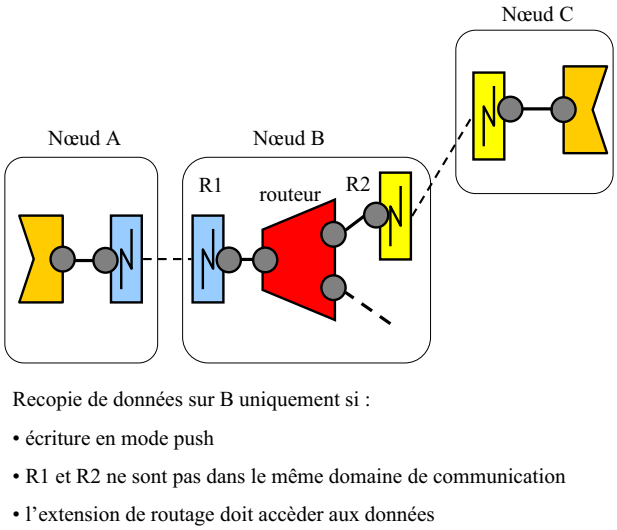


Figure 5.5 – Communication directe entre les extrémités d'un chemin Proboscis grâce à l'infrastructure IODSM

Pour clore cette section, il convient d'insister sur le fait que l'infrastructure de partage des données que nous proposons ne fournit aucun mécanisme garantissant la cohérence du contenu d'un tampon en cas d'accès concurrents. De tels mécanismes peuvent être ajoutés soit au sein d'un chemin Proboscis, soit à l'extérieur.

Dans le premier cas, (par exemple pour un SGF partagé implémenté intégralement à l'aide de modules Proboscis ou un disque virtuel réparti avec cohérence des caches), il est nécessaire de d'intégrer la prise en charge des contraintes de cohérence au sein du code d'une ou plusieurs extensions.

Dans le second cas (par exemple un ensemble de chemins émulant un SAN), la responsabilité de la gestion de la cohérence est laissée aux couches supérieures du système (qui correspondent typiquement à un SGF/SGBD partagé).

[16]Il n'est pas toujours possible d'éviter une recopie de données sur le nœud B, cela dépend du schéma de répartition employé. Par exemple, avec un schéma à la RAID 5, l'extension de dispersion a besoin d'obtenir une instance locale du BA associé à la requête pour pouvoir mettre à jour les informations de parité.

5.6 Travaux connexes

Comme nous l'avons déjà mentionné précédemment, la structure d'un assemblage de tampons est inspirée de l'infrastructure *IOLite* [167], qui vise à unifier toutes les différentes couches de tampons et de caches au sein d'un système d'exploitation afin d'augmenter les performances en éliminant toute recopie de données. L'infrastructure que nous proposons ne répond pas à cette préoccupation (elle ne permet d'éviter les recopies de données qu'en son sein). Elle pourrait néanmoins être intégrée facilement à un système reprenant le principe d'IOLite car elle repose sur les mêmes concepts de base et une interface de programmation identique.

La volonté d'éliminer les recopies de données et de fusionner les caches de données et de tampons réseau est également centrale au sein d'*Unifier* [235] et du *Network-centric buffer cache* [172]. Unifier s'intéresse principalement à minimiser le coût d'enregistrement des tampons utilisés pour des opérations de RDMA sur des réseau de type Infiniband. Le second projet cible le contexte des serveurs *pass-through*, par exemple, un serveur NFS qui n'a qu'un rôle de routeur entre un client et un ensemble de nœuds de stockage exportant des périphériques via iSCSI.

Les trois projets cités ci-dessus ont en commun une différence avec notre approche : il ne s'intéressent à la gestion des tampons d'E/S que dans le contexte local d'un nœud.

La gestion répartie de tampons de données que nous proposons est dans la lignée de nombreux travaux antérieurs sur les thèmes des mémoires partagées réparties (MPR) logicielles. Une préoccupation majeure des MPR concerne le choix et la mise en œuvre d'une politique de cohérence des données. Par opposition, notre infrastructure découple la gestion des transferts de données et celle de la cohérence et laisse ce second aspect à la discrétion des couches supérieures du système concerné. Puisque notre infrastructure est actuellement implémentée au niveau du noyau, il n'est pas possible d'utiliser les mécanismes de mémoire virtuelle pour détecter les accès à des tampons partagés. En conséquence, une application doit gérer explicitement les problèmes de cohérence ou s'appuyer éventuellement sur le support automatisé d'un compilateur et/ou d'un environnement d'exécution spécialisé[17]. Enfin, la propagation des mises à jour dans notre système et l'utilisation des couplages directs à distance via SCI sont respectivement proches des protocoles MPR *home-based* [238] et AURC (*Automatic Update Release Consistency*) [112].

Les conteneurs employés au sein du système d'exploitation réparti Gobelins/Kerrighed [180] partagent la plupart de nos objectifs. Encore une fois, notre infrastructure est moins ambitieuse car elle ne fournit pas de garanties de cohérence. En revanche, elle permet potentiellement de meilleures performances pour la transmission de données car la taille des transferts n'est pas limitée à une seule page de mémoire.

Les MPR sont généralement considérées comme une solution passant difficilement à l'échelle car le protocole de cohérence engendre de nombreuses communications et les transferts de données sont d'une granularité assez fine. Dans notre approche, la taille des transferts de données n'est bornée que par les capacités du matériel[18] et la gestion de la cohérence est laissée aux couches supérieures. Ce choix nous semble bien adapté au contexte d'application visé : les serveurs de données.

Les MPR ont pour but principal de simplifier le développement d'applications scientifiques pour grappes en fournissant un modèle de programmation et des garanties comparables à celles d'un système centralisé. En revanche, l'objectif premier de notre infrastructure est de simplifier l'adaptation d'un serveur de données à différents environnements matériels et différentes charges sans renoncer à la possibilité d'introduire des optimisations spécifiques à un contexte de déploiement précis. Les SGF et SGBD répartis assurent la cohérence des données manipulées simultanément par différents

[17] comme dans les cas des MPR dont l'unité de partage n'est pas une page de mémoire (variables ou objets partagés).
[18] et d'éventuelles limites imposées par l'interface de programmation des couches supérieures

clients (avec des mécanismes relativement efficaces car explicites dans le modèle de programmation et une granularité potentiellement supérieure pour la gestion des données). Notre infrastructure de bas niveau est destinée à être insérée en dessous ou intégrée au sein de couches de ce type et n'a donc pas besoin de fournir des mécanismes de cohérence, qui s'avéreraient souvent redondants. En contrepartie, le concept de tampons répartis que nous proposons est plus complexe à appréhender qu'une MPR et doit être utilisé avec précaution au sein d'une application.

Des projets tels que Madeleine [20] ont déjà exploré le thème des couches de communication « unificatrices » permettant de virtualiser le protocole et la technologie réseau employés, d'optimiser dynamiquement les performances en choisissant une stratégie de transfert appropriée et d'interconnecter des grappes basées sur des réseaux hétérogènes. Notre proposition partage la plupart des motivations de telles couches, mais s'en écarte sur certains points. Madeleine se positionne comme une couche intermédiaire (sous la forme d'une bibliothèque) de support à des environnements logiciels pour les applications de calcul. Notre infrastructure s'intéresse davantage aux serveurs répartis de données (fonctionnant potentiellement en mode noyau), où un besoin de souplesse important concerne le choix du moment et du nœud à l'initiative des transferts de données afin d'optimiser les échanges de données entre les extrémités d'une chaîne de traitement de requêtes selon divers critères complémentaires aux performances perçues par le client (charge du serveur et des éventuels routeurs intermédiaires).

Revenons sur ce dernier point afin de préciser les particularités de notre approche. La plupart des couches de communication unificatrices (CCU) ont été développées pour les besoins d'applications de calcul scientifique[19]. Dans ce contexte, un critère primordial concerne l'optimisation des communications (en général directes[20]) entre deux nœuds en fonction de paramètres tels que la taille des messages et les capacités du matériel. Dans ce cas, la CCU fournit une interface de communication uniforme pour les programmeurs d'application, quelles que soient la stratégie de transfert et la technologie d'interconnexion employés. Cependant, les échanges de données doivent être spécifiés explicitement dans le code applicatif (même si certains modèles de programmation, dont celui de Madeleine, permettent une certaine souplesse par rapport au moment exact d'envoi des différents tampons associés à un message).

Notre travail s'intéresse aux serveurs de stockage construits sous la forme d'un pipeline de traitement, à partir d'un assemblage de composants. De plus, un tel pipeline peut être déployé sur plus de deux nœuds. Afin de permettre l'emploi de composants génériques (ne nécessitant pas de modification de code pour une utilisation dans un assemblage particulier), il est nécessaire de fournir un modèle de programmation qui masque les communications liées aux échanges de tampons. Cette contrainte nous a conduit a découpler le transfert des descripteurs de données de celui des données proprement dites, en utilisant des techniques inspirées par le domaine des mémoires partagées réparties[21]. Ceci constitue, à nos yeux, la principale originalité des propositions exposées dans ce chapitre. Les aspects relatifs à la gestion transparente de multiples technologies d'interconnexion et de plusieurs stratégies de transfert sont moins novateurs. Nous aurions d'ailleurs pu réutiliser une CCU existante à cet effet (à condition que cette dernière soit utilisable au sein du noyau) mais nous avons préféré développer

[19] À notre connaissance, les utilisations de CCU dans le contexte des serveurs de données ont été limités à des schémas de communication semblables à ceux d'une application de calcul.

[20] Certaines CCU gèrent également les échanges de données entre deux nœuds équipés de réseaux hétérogènes, par l'intermédiaire de nœuds « passerelles ».

[21] Remarque : l'emploi de techniques proches des MPR et le fait que notre prototype ait été initialement basé sur l'interface SCI peuvent laisser penser que notre proposition est très liée à un matériel d'interconnexion capable de coupler de la mémoire distante. Nous ne pensons pas que ce soit le cas. En effet, comme nous l'avons montré, il est tout à fait envisageable de développer des stratégies de transfert basées sur d'autres mécanismes de communications offerts par la majorité des périphériques réseau tels que le RDMA ou l'envoi de paquets. En outre, notre modèle n'impose pas de restrictions sur la manière dont le contenu d'un assemblage de tampons doit être transmis (un ou plusieurs envois).

notre propre infrastructure afin de simplifier l'intégration avec le code de gestion des assemblages de tampons et d'éviter un empilement excessif de couches plus ou moins redondantes.

Nos travaux rejoignent aussi la proposition récente de « Datamover architecture » (DA) [46] développée au sein de la pile protocolaire iSCSI. La spécification DA définit une interface abstraite pour les transferts de données dans le cadre des échanges entre un client et un serveur iSCSI. Différentes implémentations de l'interface peuvent être employées en fonction du matériel considéré : pile logicielle TCP, accélérateur matériel TCP, transferts par RDMA [125], etc.

Par rapport à l'interface DA, notre proposition se situe à un plus haut niveau logique, elle inclut un modèle de programmation pour abstraire les données au niveau applicatif, est indépendante des protocoles des couches supérieures et permet dans certains cas, des transferts directs entre les extrémités d'une chaîne de nœuds de routage.

Enfin, notre infrastructure est également influencée par les concepts de gestion globale de la mémoire d'une grappe [69] et de caches coopératifs [60] et peut servir de support à de telles fonctionnalités. Un autre aspect associé à la gestion globale de la mémoire concerne le contrôle des ressources au niveau de nœuds serveurs impliqués dans des requêtes d'E/S. Sur ce point, notre démarche rejoint les travaux tels que *disk-directed I/O* [126] et *server-directed collective I/O* [198]. Là encore, notre approche pourrait servir de brique de base pour la construction de tels systèmes, et permettre une adaptation aisée à différentes technologies d'interconnexion ainsi qu'à des charges d'E/S variées.

5.7 Bilan

Dans ce chapitre, nous avons proposé une infrastructure qui permet aux nœuds d'une grappe de partager des tampons en mémoire de manière efficace. Les transferts de données sont considérés comme des aspects non-fonctionnels qui peuvent être configurés, de façon statique ou dynamique, indépendamment de l'application qui les effectue, en fonction de l'environnement de déploiement (capacité du matériel, topologie physique du réseau, topologie logique de l'application) et des contraintes d'exploitation (telles que des critères de charge sur les différents nœuds).

Le modèle de programmation proposé n'impose pas de restrictions fortes sur les stratégies de transfert qu'il est possible d'utiliser. Nous en avons présenté plusieurs exemples dans le contexte des interfaces SCI et TCP/IP. Certaines d'entre elles permettent d'éliminer les recopies de données intermédiaires sur des nœuds ne jouant qu'un rôle de routeur et d'obtenir ainsi un transfert direct entre une source et un puits de données.

Chapitre 6

Infrastructure d'administration et mécanismes de reconfiguration dynamique

Sommaire

Le chapitre 3 a motivé et détaillé les besoins des systèmes de stockage réparti en matière de reconfiguration dynamique. Ce chapitre présente nos propositions à ce sujet, développées dans le contexte de Proboscis. Au préalable, nous décrivons l'infrastructure d'administration sur laquelle reposent les mécanismes de déploiement et de reconfiguration.

Le reste est organisé de la façon suivante. La section 6.1 présente l'infrastructure d'administration. Les sections 6.2, 6.3 et 6.4 abordent respectivement les reconfigurations de code, des paramètres d'un composant et de la structure du système de stockage. Des exemples de protocoles de reconfiguration sont décrits en 6.5. Enfin, les fondements pour le développement de systèmes de stockage autonomes sont explicitées en section 6.6.

6.1 Infrastructure d'administration

Le système d'administration a plusieurs responsabilités :

- simplifier la tâche d'un administrateur en lui fournissant une console centrale pour la supervision et le contrôle du système de stockage ;
- permettre la distribution de nouveaux modules de code (ou de mises à jour d'un module déjà déployé) sur un ensemble de noeuds ;
- surveiller les nœuds de la grappe pour détecter d'éventuelles pannes ;
- fournir le service de nommage (dont l'utilisation a été décrite en 4.4.3) ;
- maintenir des informations sur l'état courant (en termes de configuration) du (ou des) systèmes de stockage actuellement déployé(s) ;
- garantir la cohérence des reconfigurations dynamiques en synchronisant des opérations concurrentes.

Cette session décrit de façon succincte comment ces différents objectifs sont pris en compte.

L'infrastructure d'administration est basée sur un serveur d'état central (SEC par la suite), capable de communiquer directement avec chacun des nœuds de la grappe. Un démon d'administration (`pad` — *Proboscis Administration Daemon*) est déployé sur chaque nœud de la grappe. Il permet ainsi une communication bidirectionnelle et notamment :

- au SEC d'envoyer des messages d'administration à l'infrastructure Proboscis locale du nœud (un message est traduit en un appel système `ioctl` à l'infrastructure) ;
- à l'infrastructure locale d'acquitter un message d'administration ;
- à toute instance d'extension de contacter le SEC (via un mandataire fourni par l'infrastructure locale).

La détection de la panne d'un nœud est basée sur un mécanisme de *heartbeat* coordonné par le SEC. Plusieurs services déployés sur une grappe s'appuient généralement sur cette fonction ; dans ce cas, le SEC peut bien sûr souscrire au service d'appartenance existant plutôt que d'en réimplanter un et causer des communications redondantes.

Le SEC intègre le service de nommage. Ce dernier repose sur une base de données modifiable dynamiquement par l'administrateur. Une entrée dans la base associe un nom à la description textuelle d'un chemin et indique si celui-ci est libre ou fixé. Dans le second cas, l'entrée précise également le type d'extension réseau (et les paramètres appropriés) à utiliser si le chemin considéré n'est pas fixé sur le nœud à l'initiative de la résolution de nom.

Le SEC maintient également, comme son nom l'indique, des informations sur l'état actuel du système de stockage en cours de fonctionnement. Ces informations sont stockées sous la forme de *squelettes* (cf. 4.4.2) et regroupent notamment la structure des chemins déployés, et pour chaque instance, son identifiant unique et la valeur courante des attributs qu'elle exporte[1]. Chaque opération de création, destruction ou reconfiguration d'un chemin doit mettre à jour l'état correspondant au niveau du SEC. Ces mises à jour sont garanties par l'infrastructure locale à chaque nœud : (1) si l'opération a été démarrée par le SEC via un message d'administration, alors le squelette mis à jour est joint à l'acquittement du message, (2) si l'opération a été au contraire initiée de façon locale, l'infrastructure locale prend l'initiative de notifier le résultat au SEC.

Le SEC gère aussi un ensemble de verrous pour synchroniser les modifications concurrentes d'une même portion de chemin. Ce mécanisme permet également de rendre atomique une reconfiguration

[1] Une extension capable de modifier elle-même la valeur d'un attribut qu'elle exporte peut déclarer celui-ci comme étant « volatil ». En conséquence, ses mises à jour ne seront pas transmises au SEC. Ceci permet d'éviter des communications superflues lorsqu'une instance ajuste très fréquemment ses paramètres opérationnels.

composée de plusieurs étapes élémentaires. Il est possible de verrouiller un chemin complet ou seulement une partie d'un chemin. Une demande de verrous échoue si au moins l'une des instances visées est déjà verrouillée.

Le fait d'employer un serveur central est naturellement préoccupant en termes de passage à l'échelle et de tolérance aux pannes. Nous pensons cependant que cette solution est acceptable dans le contexte qui nous intéresse.

Tout d'abord, nous faisons l'hypothèse que le trafic réseau lié aux messages d'administration et au protocole d'appartenance transite sur un réseau dédié à l'administration de la grappe ; les communications applicatives et les transferts de données ne sont donc pas pénalisés[2]. Des travaux antérieurs [226] montrent que les mécanismes d'appartenance habituels passent à l'échelle au sein d'une grappe, pour peu que le réseau sous-jacent offre un mécanisme de diffusion (ce qui est le cas pour Ethernet).

Le SEC est uniquement impliqué dans les tâches d'administration et non dans les E/S. Le nombre de chemins déployés au sein d'une grappe évolue peu au cours du temps et les opérations de reconfiguration sont très ponctuelles (et peuvent généralement être échelonnées), les risques de surcharge du serveur sont donc faibles.

L'argument précédent sur la fréquence des modifications permet également de traiter simplement les problèmes de défaillances. L'emploi d'un nœud de secours synchronisé avec le SEC via un protocole maître/esclave et un accès partagé à une base d'informations persistante semble suffisant pour faire face aux dysfonctionnements du serveur central. Avec une telle configuration, la reprise sur panne est rapide et affecte uniquement la latence des opérations d'administration (les opérations d'E/S n'impliquent pas le SEC).

6.2 Ajout d'extension et mise à jour de code

Cette section commence par décrire comment de nouveaux types d'extensions peuvent être déployés de manière dynamique au sein d'une grappe. Dans un second temps, elle expose le mécanisme permettant de mettre à jour le code d'extensions existantes.

6.2.1 Déploiement de nouveaux types d'extensions

Chaque extension Proboscis se présente sous la forme d'un module de code chargeable par le noyau du système d'exploitation d'un nœud (comme un pilote de périphérique « classique »). Lors du chargement au sein du noyau, l'extension s'enregistre auprès de l'infrastructure Proboscis locale (elle-même implémentée sous la forme d'un module noyau), qui maintient une liste des extensions enregistrées. Si l'infrastructure Proboscis n'est pas présente, le chargement de l'extension échoue.

Le SEC peut interroger l'infrastructure déployée sur chacun des nœuds afin de déterminer la liste courante des extensions installées (c'est-à-dire dont le module de code est disponible dans l'arborescence locale) et déployées (c'est-à-dire chargées au sein du système). Ceci permet un processus d'amorçage : lors du (re)démarrage du serveur, celui-ci peut (re)construire une base de données cohérente sur l'état courant de la configuration logicielle des nœuds.

Le serveur peut aussi envoyer à un nœud, via un message d'administration :

- un module de code associé à un nouveau type d'extension[3] ;

[2] Cette hypothèse s'avère réaliste au vu des configurations actuelles. La plupart des grappes sont désormais équipées d'un réseau (généralement de type FastEthernet) dédié à leur administration et à leur supervision.

[3] Plutôt que d'envoyer une copie du module de code sur chaque nœud, il serait plus simple d'envoyer uniquement une référence vers des fichiers stockés dans un espace accessible par tous les nœuds. Notre proposition évite cette hypothèse

- l'ordre de charger un module au sein du système d'exploitation (et par conséquent au sein de l'infrastructure Proboscis) ;
- l'ordre de décharger et désinstaller un module (uniquement s'il n'existe aucune instance associée au type d'extension considéré).

Ainsi, il est possible de modifier la base d'extensions disponibles sur chaque nœud. Dans la plupart des cas, il est souhaitable d'avoir une configuration homogène pour tous les nœuds d'une grappe mais certains usages peuvent nécessiter davantage de flexibilité (grappe hétérogène dont certains nœuds ont des ressources mémoire restreintes et ne nécessitent qu'un petit ensemble d'extensions).

6.2.2 Mise à jour du code d'une extension

Nous nous intéressons maintenant à la mise à jour du code d'une extension dont le module binaire associé a déjà été distribué au sein de la grappe. Pour cela, un numéro de version est associée à une extension et le SEC maintient un inventaire des versions installées sur chaque nœud.

Dans le cas le plus simple, il n'existe aucune instance associée au type d'extension considéré. Il suffit alors, pour chaque nœud concerné, de décharger et désinstaller l'ancienne version du module correspondant puis d'installer et charger la nouvelle version du module.

Le problème du remplacement « à chaud » d'un composant logiciel a déjà été l'objet de nombreux travaux de recherche dans des domaines d'application variés ; les plus proches de nos préoccupations ont été menés dans le cadre du système d'exploitation K42 [207, 24]. Les mises à jour considérées ont les objectifs suivants, que nous partageons :

- la suppression de trous de sécurité, de problèmes de performances (dues à une mauvaise implémentation) et de bogues[4] ;
- l'instrumentation dynamique du code (prélèvement d'informations précises uniquement lorsque cela est nécessaire) ;
- la capacité de jongler entre plusieurs implémentations optimisées pour différentes conditions opérationnelles.

Une limitation courante des mécanismes de K42 est qu'il est impossible de modifier les interfaces d'un composant lors d'un changement de version, ni d'effectuer la mise à jour de plusieurs composants de façon synchronisée (ce qui peut justement être utile si l'interface qui les relie doit être changée). Ces limitations ne sont pas rédhibitoires dans notre cas car, dans le modèle de composition de Proboscis, les interfaces d'un composant ne spécifient que le niveau d'abstraction utilisé pour la manipulation des données, les interactions entre composants sont spécifiées par les types de commandes qui transitent d'un port à l'autre. De plus, nous considérons que les mises à jour de code qui induisent une profonde modification du comportement d'un composant (en termes d'interaction avec ses pairs) relèvent davantage de l'introduction d'un nouveau type d'extension, et donc d'un changement de structure du système. Ce type de reconfiguration est géré de manière distincte (voir 6.4).

Dans [207], quatre capacités nécessaires au support du remplacement dynamique de composants sont identifiées. Le système doit être capable :

---

car l'objectif principal de Proboscis est justement la construction de systèmes de stockage partagés pour grappes ! Une telle simplification pourrait cependant être mise en œuvre facilement en employant un serveur NFS au niveau du SEC.

[4] Dans le cas des bogues, une mise à jour n'a, bien évidemment, qu'un rôle préventif. Si un bogue a déjà altéré l'état du système, cette mesure est insuffisante.

1. d'identifier et d'encapsuler le code et les données associés à une instance de composant ;
2. d'amener une instance dans un état passif, c'est-à-dire qu'il n'y ait plus d'entité active qui possède une référence vers son code ou ses données ;
3. de traduire et transférer l'état de l'ancienne instance vers celui de la nouvelle ;
4. de modifier toutes les références vers l'instance remplacée.

Nous allons maintenant expliquer comment ces critères sont respectés au sein de l'infrastructure Proboscis.

1. Le modèle de composant utilisé par Proboscis satisfait intrinsèquement le premier critère.
2. Le fait d'utiliser un modèle de programmation par événements combiné à l'emploi d'un seul thread d'exécution permet de déterminer facilement[5] lorsqu'une instance n'est pas active et de bloquer temporairement les appels de méthodes sur celle-ci.
3. La gestion du transfert d'état est à la charge du programmeur. Pour chaque version d'une extension, il est nécessaire de fournir un couple de méthodes permettant de convertir l'état d'une instance vers une représentation canonique et vice versa. Par ailleurs, ces méthodes sont aussi utilisées pour la (dé)sérialisation préalable (postérieure) à la migration d'une instance (voir 6.4.2.3).

   La représentation sous forme canonique de l'état d'une instance correspond à une chaîne de caractères. Ce choix est moins efficace qu'une représentation binaire mais il permet d'intégrer l'état courant d'un chemin au sein d'une description textuelle. Il est ainsi possible de redéployer un chemin dont l'état a été capturé. L'utilisation de ce mécanisme est détaillée en 6.4.2.2.
4. Le dernier critère est respecté grâce à l'emploi de pointeurs d'indirection, à la fois pour les méthodes associées à une instance et pour les liaisons entre instances.

Nous allons à présent décrire, de façon concrète, les étapes de la mise à jour du code associé à une extension sur un nœud N. Le SEC commence par envoyer la nouvelle version du module de code sur N. Ce module est ensuite chargé par l'infrastructure Proboscis. Plusieurs versions d'un même module peuvent cohabiter au sein de l'infrastructure (prévue à cet effet) et du système d'exploitation sous-jacent car les symboles exportés par chaque version du code sont différents (ils incluent un numéro de version grâce à l'emploi de macros dans le code source).

Dans un second temps, le SEC envoie un message d'administration pour démarrer l'échange de versions. L'opération peut viser toutes les instances du type d'extension concerné déployées sur N (cas le plus fréquent) ou seulement une instance, désignée par son identifiant unique (cela peut être utile pour l'instrumentation fine d'une portion précise de chemin). À la réception du message d'administration, le démon `pad` transmet une demande de mise à jour à l'infrastructure d'exécution (cette demande ne se traduit pas par la création d'une requête qui traverse un chemin mais par la demande d'ordonnancement d'une tâche indépendante). La tâche en question consiste simplement à :

1. identifier toutes les instances concernées par la mise à jour (ceci est possible car l'infrastructure joue le rôle d'« usine à composants » et répertorie toutes les instances déployées pour un type d'extension donné) ;
2. pour chaque instance, effectuer la mise à jour :
   - récupérer l'état de l'ancienne version sous forme canonique ;
   - créer la nouvelle version de l'instance en récupérant l'état sérialisé ;
   - mettre à jour les liaisons pointant vers l'ancienne instance ;

[5] Et en particulier, sans recourir au protocole à trois phases décrit dans [207].

- détruire l'ancienne instance.

3. si nécessaire (mise à jour permanente pour toutes les instances), mettre à jour l'usine à composants pour utiliser la nouvelle version par défaut.

Enfin, le SEC peut éventuellement envoyer un dernier message pour décharger et désinstaller l'ancienne version.

Pour terminer, il convient d'expliciter les limites de notre approche. Comme indiqué précédemment, le mécanisme proposé ne permet pas de modifier les interfaces d'un composant mais uniquement son implémentation. Cette restriction peut être contournée en assimilant la nouvelle version à un nouveau type d'extension et en effectuant une reconfiguration de la structure des chemins concernés.

Le support des mécanismes de transfert d'état implique la participation du programmeur de chaque type d'extension. Le volume de code à produire est cependant limité car une extension est généralement associée à une fonctionnalité élémentaire. En outre, des travaux antérieurs laissent envisager la possibilité de générer automatiquement les méthodes de transfert d'état dans des cas simples [97].

Le mécanisme de transfert d'état peut également poser des problèmes en termes de consommation de ressources et de performances. Dans certains cas, l'état englobé par un composant peut être volumineux (par exemple pour une extension qui fournit le rôle d'un cache) et/ou coûteux à recopier/réallouer (nombreuses structures de données interconnectées, etc.). Si l'ancienne et la nouvelle version utilisent exactement les mêmes définitions de structures de données (ce qui est le cas pour la plupart des mises à jour, qui consistent uniquement à ajouter ou modifier quelques instructions) alors l'étape de transfert d'état peut être optimisée : au lieu de sérialiser le « contenu » de l'état sous forme canonique, l'ancienne instance peut communiquer un ou plusieurs pointeurs à la nouvelle. Ceci n'est toutefois possible qu'à deux conditions complémentaires :

- l'infrastructure doit fournir un mécanisme de négociation du protocole de transfert, qui permet de déterminer dans quelles situations (et de quelle manière) l'état peut être transféré directement d'une instance à l'autre [207] ;
- le code de l'ancienne version doit prévoir explicitement ce genre de situation : lorsque l'état a été directement transmis à la nouvelle instance, le code associé à la destruction de l'ancienne instance doit veiller à ne pas détruire les structures de données concernées.

Le programmeur est également responsable de fournir une fonctionnalité équivalente d'une version à l'autre. L'infrastructure n'effectue aucune vérification en ce sens[6]. Une fois de plus, compte tenu du niveau de granularité associé à une extension et du fait que les modifications apportées d'une version à l'autre sont généralement mineures, cette hypothèse semble acceptable.

Une mise à jour est appliquée immédiatement, ou du moins dès que la tâche correspondante est ordonnancée par l'infrastructure d'exécution. Dans certaines situations, il serait souhaitable de différer la mise à jour tout en garantissant qu'elle sera effectuée, au plus tard, juste avant le prochain appel au(x) composant(s) visé(s). Ainsi, les performances du système de stockage ne seraient affectées par la mise à jour qu'au « dernier moment », lorsque celle-ci devient véritablement nécessaire. Il n'y a pas d'obstacle théorique à l'implémentation de cette caractéristique mais cela requiert d'étendre les capacités d'interposition de l'infrastructure d'exécution. De plus, la phase de recherche des composants à mettre à jour ne peut être retardée (il est nécessaire d'identifier les composants vers lesquels les appels devront être interceptés) ; l'idée de différer les modifications n'est donc intéressante que pour les composants dont la phase de transfert d'état est très coûteuse.

[6] Nous rejoignons à ce sujet les arguments développés en 7.1 dans [207].

6.3 Modification des paramètres du système

Le second type de reconfiguration que nous considérons permet d'influer sur le comportement d'une instance d'extension en modifiant la valeur d'un ou plusieurs des attributs qu'elle exporte. Par exemple, il peut s'avérer souhaitable de changer dynamiquement la taille de la zone de mémoire allouée à une instance de cache, la politique d'ordonnancement des requêtes utilisée par une instance d'agrégation[7], etc.

Le mécanisme repose sur l'envoi d'une requête générique (de type `UPDATE_ATTR`) à l'instance concernée. A l'instar des mises à jour de code, la reconfiguration d'un paramètre d'une instance nécessite le support du programmeur de chaque extension. Le développeur doit fournir un traitant principal pour les requêtes de type `UPDATE_ATTR` ainsi qu'un traitant spécialisé pour chaque paramètre reconfigurable. Le traitant principal ne sert qu'à déterminer le traitant spécialisé à appeler.

Dans certains cas, la mise en œuvre du traitant spécifique est très simple[8]. Si un paramètre correspond uniquement à une valeur numérique réévaluée systématiquement par le code de l'extension, le traitant spécialisé consiste seulement à modifier la valeur de la variable correspondante[9].

En revanche, dans d'autres cas, le traitant est plus complexe. Ainsi la modification de la taille d'un cache va entraîner des actions d'allocation/libération/recopie de mémoire, d'éventuelles mises à jour de variables internes, etc. Quoi qu'il en soit, l'implémentation d'une telle procédure est simplifiée par le fait que la reconfiguration est traitée comme une requête quelconque : il n'y a pas de problème de concurrence avec une autre requête ou un événement envoyés à l'instance concernée. Il est cependant possible qu'au moment où elle reçoit la requête, l'instance ne soit pas dans un état capable de supporter la reconfiguration. Dans ce cas, l'instance peut différer le traitement de la commande, de deux manières. La première consiste à demander l'ordonnancement (retardé) d'un événement. Cette méthode est simple mais ne fournit pas de garantie déterministe sur le succès final de la reconfiguration. Le programmeur peut aussi gérer explicitement une file d'attente pour les commandes de type `UPDATE_ATTR` et déclencher leur traitement (en ordonnançant un événement prioritaire) lorsqu'un état propice est atteint.

Pour compléter ces explications, nous décrivons maintenant la séquence complète associée à la reconfiguration d'un paramètre d'une instance.

1. Le SEC envoie un message d'administration au nœud N où débute le chemin qui contient l'instance concernée (mais l'instance peut être déployée sur un autre nœud si le chemin est réparti)[10]. Le message contient l'identifiant de l'instance ainsi que deux chaînes de caractères. La première correspond au nom de l'attribut à modifier, la seconde à la nouvelle valeur de l'attribut (la désignation est basée sur la même description textuelle que celle utilisée pour la création d'un chemin).
2. Sur le nœud N, le démon d'administration crée un commande `UPDATE_ATTR` qui contient les mêmes informations que le message initial. Le routage de la commande est calculé à la source en fonction de l'identifiant de l'instance cible (les informations de routage sont fournies par le

[7] en supposant que le code existant permet de choisir entre différentes stratégies ou de pondérer explicitement la priorité associée à chaque port ; dans le cas contraire, la reconfiguration relève d'une mise à jour de code.

[8] et le traitant pourrait même être généré automatiquement sans grande difficulté.

[9] Exemple typique de paramètre réévalué de manière systématique : la fréquence à laquelle une extension « autonome » ordonne des événements pour cadencer son activité (fréquence de synchronisation d'un cache, etc.).

[10] Si l'instance visée correspond à une extension d'agrégation, ou se situe en aval d'une instance d'agrégation, il existe plusieurs possibilités pour le nœud de départ. Le choix peut être aléatoire ou basé sur des critères de charge.

message d'administration émanant du SEC, qui a toujours une vue à jour de la structure des chemins déployés au sein de la grappe).

3. La commande traverse le chemin jusqu'à l'instance cible.
4. La commande est traitée par l'instance comme expliqué ci-dessus. Elle n'est acquittée qu'au moment où la reconfiguration a été effectuée (ou a échoué).
5. Lorsque la commande acquittée revient à l'origine du chemin, elle est détruite et le message d'administration est acquitté auprès du SEC, qui met à jour les informations qu'il maintient sur le chemin concerné.

Tous les attributs ne sont pas modifiables dynamiquement ; nous donnons ci-dessous quelques contre-exemples représentatifs.

Les attributs d'une extension d'extrémité servent (pour la plupart) à spécifier le point d'ancrage du chemin par rapport aux structures de données du système d'exploitation. Par exemple, une extension racine de niveau bloc est associée à un identifiant de périphérique logique au sein du système. Ce paramètre ne peut pas être modifié dynamiquement car il n'est pas interne à l'infrastructure Proboscis : sa valeur à une incidence sur le bon fonctionnement des couches supérieures (système d'exploitation et applications) qui utilisent le chemin.

Au sein d'une extension réseau, il n'est pas possible de modifier directement le périphérique de communication sous-jacent, ni l'adresse associée au périphérique. Ceci tient au fait qu'une extension réseau est en fait constituée de deux talons déployés sur des nœuds distincts. Il est cependant possible d'aboutir au résultat désiré par le biais d'une reconfiguration structurelle, le procédé est décrit en 6.5.1.

6.4 Modification de la structure du système

Ce type de reconfiguration est primordial pour qu'un système de stockage puisse satisfaire des contraintes de haute disponibilité. Il repose sur deux mécanismes de base : l'insertion ou la suppression d'extensions au sein d'un chemin. Cette section commence par présenter les deux mécanismes de base pour les reconfigurations structurelles puis montre comment ceux-ci peuvent être exploités pour fournir des fonctionnalités plus avancées.

6.4.1 Mécanismes de base

6.4.1.1 Insertion d'extensions

Ce premier mécanisme de reconfiguration structurelle permet d'étendre la composition d'un chemin déjà existant par l'insertion d'un nouveau chemin (constitué d'une ou plusieurs extensions). L'insertion peut avoir lieu :

- au milieu du chemin initial, en remplaçant une liaison entre deux instances par le nouveau chemin ;
- à l'extrémité du chemin initial (au niveau d'une extension de dispersion).

Ces deux modes d'insertion sont désignés ci-dessous par les expressions respectives d'*insertion intermédiaire* et d'*insertion terminale* et sont illustrés sur la figure 6.1 (les instances insérées sont grisées).

Remarque : La (dé)connexion d'un client auprès d'une extension d'agrégation déjà déployée (comme décrit en 4.4.3) n'est pas considérée comme une opération de reconfiguration car elle peut être effectuée uniquement avec une commande de création (destruction).

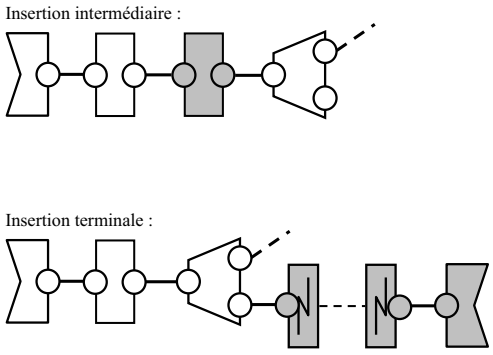


Figure 6.1 – Modes d'insertion d'un chemin

Nous pouvons maintenant décrire la séquence caractéristique d'une opération d'insertion.

1. Comme pour la modification des paramètres d'une extension (6.3), le SEC envoie un message d'administration au nœud N où débute le chemin visé par l'opération d'insertion. Le message inclut notamment l'identificateur de l'instance où doit démarrer le processus d'insertion ainsi que la description textuelle du chemin à insérer.
2. Sur le nœud N, le démon d'administration crée une commande `INSERT` qui contient les mêmes informations que le message initial. Le routage de la commande est déterminé à la source.
3. La commande traverse le chemin jusqu'à l'instance cible (à partir de laquelle le nouveau chemin va être inséré), dénommée A par la suite.
4. Lorsque A reçoit la commande, elle n'a rien à faire de particulier. Il lui suffit d'appeler le traitant par défaut, qui lui-même appelle `prob_cmd_issue`. L'infrastructure d'exécution détecte alors qu'il est temps d'amorcer le processus d'insertion.
5. Le processus d'insertion est assuré de s'exécuter sans concurrence avec d'autres opérations d'administration sur le même chemin grâce aux verrous gérés par le SEC. Cependant, l'insertion est effectuée de manière concurrente avec le flux de requêtes d'E/S[11] qui traverse le chemin. Pour cela, l'infrastructure utilise une barrière de synchronisation. Elle bloque l'émission de nouvelles requêtes vers l'instance A et autorise uniquement l'acquittement des commandes en cours. Ainsi, il est possible d'atteindre un état tel qu'il n'existe plus aucune requête ayant traversé A encore en transit au sein du chemin. Lorsque ce critère est satisfait, l'infrastructure peut effectivement traiter la requête d'insertion qu'elle avait mise en attente.
6. L'infrastructure crée une commande `CREATE` associée au chemin à insérer et l'émet à partir de A, sur le port concerné[12].
7. Le processus de création est relativement analogue a celui décrit en 4.4.2. En particulier, la phase « aller » sert à créer les instances et allouer les ressources associées alors que la phase

[11] Nous désignons ici par « requêtes d'E/S » tous les types de requêtes utilisés pour la gestion des échanges de données entre nœuds : ordres de lecture et d'écriture, bien sûr, mais aussi les commandes de synchronisation (gestion des verrous associés aux données, invalidation de caches), etc.

[12] Si l'instance A correspond à une extension de dispersion, il est nécessaire de préciser à partir de quel port doit commencer le processus d'insertion. Un paramètre de la commande `INSERT` est prévu à cet effet. Si l'insertion a lieu entre deux instances, le numéro de port doit correspondre à un port existant. Au contraire, si l'opération consiste à greffer une nouvelle branche à une instance de dispersion, le numéro de port doit être libre (il est possible de laisser l'infrastructure choisir le numéro de port à utiliser).

d'acquittement met à jour les liaisons entre instances. Ainsi, la structure du chemin initial n'est modifiée que si toutes les nouvelles instances ont pu être initialisées correctement.

Dans le cas d'une insertion intermédiaire, l'infrastructure intercepte la requête de création au niveau de l'appel à `prob_cmd_issue` depuis la dernière instance insérée, effectue la liaison avec l'instance B (liée à A jusqu'alors) et acquitte la commande.

8. Lorsque la commande de création acquittée arrive au niveau de A, elle est interceptée par l'infrastructure au niveau de l'appel à `prob_cmd_done`.

   Pour chaque type d'extension, il est possible de préciser si la commande de création acquittée doit être transmise à l'extension (c'est à dire traitée par le code spécifique de l'extension, via un appel à `done_fn`) ou gérée de façon transparente par l'infrastructure[13]. De manière générale, il est souvent souhaitable :

   - pour une insertion intermédiaire, de ne pas transmettre la commande à l'extension. Ainsi, l'opération de reconfiguration est transparente et ne nécessite aucune prise en charge au niveau du code de l'extension A ;
   - pour une insertion terminale, de transmettre la commande à l'extension. La reconfiguration n'est alors pas transparente vis à vis de l'extension A mais il est peut justement être nécessaire de l'avertir des modifications effectuées en aval d'elle. Par exemple, une extension de duplication doit être avertie de l'ajout ou du retrait d'un sous-chemin (ce point est illustré plus en détails en 6.5.2).

     Que la commande de création acquittée soit transmise à l'extension A ou non, celle-ci est, au final, toujours interceptée par l'infrastructure lors d'un appel à `prob_cmd_done` dans le contexte de A. Elle est alors détruite et la commande `INSERT` qui avait engendré le processus de création est acquittée.
   - L'acquittement de la commande d'insertion débloque les requêtes d'E/S mises en attente au niveau de l'instance A.
   - Lorsque la commande d'insertion a remonté l'intégralité du chemin, un message d'administration est envoyé au SEC pour acquitter l'opération de reconfiguration, relâcher le verrou associé et mettre à jour les méta-données centrales associées à la structure du chemin.

Ce mécanisme de reconfiguration implique, comme nous l'avons vu, une suspension temporaire des opérations d'E/S associées au chemin visé. Cependant, cette interruption du trafic est transparente (au niveau fonctionnel — les performances peuvent, quant à elles, être dégradées pendant cet intervalle de temps) pour les extensions située en amont du point de reconfiguration et, *a fortiori*, pour les couches supérieures du système qui s'appuient sur le chemin Proboscis. En outre, la durée de la suspension est généralement brève et n'impose pas un surcoût inacceptable (cf. 7.6.2).

Nous avons opté pour cette méthode en raison de sa relative simplicité, à la fois pour la mise en œuvre des mécanismes de reconfiguration ainsi que pour les programmeurs d'extensions. Sans le recours à une barrière de synchronisation, la tâche d'un programmeur serait complexe parce qu'il faudrait gérer le cas d'une commande susceptible de traverser un chemin modifié entre son trajet aller et sa phase de retour. Ceci n'est pas un problème trivial car une commande Proboscis est associée à un état (dont une pile, cf. 4.3.1.1, que chaque instance peut modifier). Garantir la cohérence d'une commande dans ce genre de situation s'avérerait fastidieux, avec de surcroît, un gain de performances

[13] L'infrastructure fournit l'interface nécessaire à un programmeur d'extension pour déterminer si une commande de création (ou destruction) d'un chemin est associée à un processus de déploiement « normal » ou à une opération de reconfiguration.

potentiellement modeste à la clé. Au contraire, avec la méthode proposée, le code d'une majorité d'extensions ne nécessite aucune prise en charge explicite des opérations de reconfiguration structurelle. La seule exception concerne les extensions de dispersion, pour lesquelles il est difficilement envisageable de masquer une opération d'insertion terminale.

Pour finir, il convient d'ajouter deux précisions importantes.

Tout d'abord, une opération de reconfiguration structurelle peut également être démarrée à l'initiative d'une instance. Dans ce cas, c'est au code de celle-ci qu'il revient :

- de contacter le SEC pour obtenir un verrou sur le chemin concerné ;
- de gérer l'allocation, l'émission, la récupération et la destruction de la commande `INSERT` ;
- de libérer le verrou et de mettre à jour les méta-données centrales auprès du SEC.

Cette situation est illustrée dans l'exemple présenté en 6.5.2.

Enfin, une opération d'*insertion intermédiaire* telle que nous l'avons décrite ci-dessus n'a de sens que dans le contexte local d'un nœud (c'est-à-dire qu'elle n'est pas adaptée à l'insertion d'un chemin comprenant une ou plusieurs extensions réseau). Une modification de la structure d'un chemin opérée à un niveau intermédiaire et ayant des répercussions sur plusieurs nœuds doit être gérée par le biais d'un protocole de reconfiguration plus élaboré (qui repose néanmoins sur les mécanismes de base d'insertion et de retrait d'extensions). Cet aspect sera explicité au fil des sections suivantes. En revanche, une opération d'*insertion terminale* peut sans problème entraîner le déploiement de nouvelles instances sur différents nœuds.

6.4.1.2 Retrait d'extensions

Le retrait d'une ou plusieurs instances au sein d'un chemin est basé sur des principes similaires à ceux d'une insertion, en remplaçant respectivement les commandes `INSERT` et `CREATE` par `REMOVE` et `DESTROY`.

Une difficulté supplémentaire tient au fait que les instances visées doivent être dans un état[14] compatible avec leur retrait « à chaud ». La phase « aller » de la commande de destruction permet de vérifier que toutes les instances concernées par le retrait sont effectivement dans un tel état. Dans le cas contraire, la tentative de retrait échoue et la structure initiale du chemin est préservée. Le signalement d'un état incompatible avec le retrait est à la charge du programmeur de chaque type d'extension, via le positionnement d'un fanion (associé à chaque instance) prévu à cet effet.

Pour la plupart des configurations, l'emploi de la technique de synchronisation décrite en 6.4.1.1 (blocage des nouvelles commandes et attente du retour des commandes en cours) suffit pour atteindre un état compatible avec le retrait. Sinon, le programmeur est libre de gérer explicitement un protocole de reconfiguration plus complexe. Afin de ne pas introduire de dépendances entre les différents types d'extensions (et ainsi préserver la modularité du canevas), il est conseillé d'implémenter, autant que possible, le protocole de reconfiguration à l'extérieur des extensions « fonctionnelles » et d'interposer temporairement une ou plusieurs extensions spécialisées pour la reconfiguration considérée.

6.4.2 Utilisation des mécanismes de base

Les deux mécanismes de base pour les reconfigurations structurelles ayant été décrits, nous pouvons maintenant présenter comment ceux-ci peuvent être exploités pour répondre à des besoins précis.

[14] Nous incluons ici, dans cette notion d'état associé à une instance, l'état des threads assistants qu'elle a déployés.

6.4.2.1 Interposition

Un premier emploi des fonctionnalités d'ajout/retrait d'instances concerne les possibilités d'interposition, c'est-à-dire d'interception des appels à un composant. Cette technique peut notamment être utilisée pour :

- l'**insertion de sondes** au sein d'un chemin afin de prélever des statistiques sur les performances du système, voire surveiller son bon fonctionnement ;
- l'insertion d'un ou plusieurs composants spécialisés implémentant un **protocole de reconfiguration**. Ces composants ont pour mission de bloquer temporairement certaines commandes, de dérouter progressivement des commandes vers un autre chemin ou encore de transférer des données d'un emplacement à un autre.

Les composants insérés ne sont généralement utiles que dans un intervalle de temps limité. Il n'est souvent pas nécessaire d'observer continuellement le fonctionnement du système à un grain très fin et les extensions de reconfiguration deviennent inutiles une fois le protocole achevé. Les instances interposées peuvent alors être retirées du chemin afin d'améliorer les performances du système de stockage.

6.4.2.2 Reconstruction d'un chemin

Il est parfois nécessaire de détruire puis reconstruire un chemin pour lui appliquer des modifications. Ainsi, si l'on souhaite modifier la façon dont différents types d'extensions interagissent, ceci peut induire une modification des champs associés à certains des types de commandes qu'ils échangent. Dans ce cas, une approche uniquement basée sur des modifications du code (cf. 6.2.2) n'est pas envisageable car il n'est possible de mettre à jour qu'un seul type d'extension à la fois[15].

Pour résoudre ce problème, nous proposons la technique décrite ci-dessous. Elle est basée sur un mécanisme lié aux fonctions d'insertion/retrait que nous n'avons pas encore détaillées. Lorsqu'une ou plusieurs instances sont retirées d'un chemin, il est possible de profiter du processus de destruction pour capturer l'état des instances visées. L'état de la portion de chemin détruite est extrait grâce à la fonction de transfert vers une représentation canonique, que chaque extension doit fournir. Cette représentation canonique correspond à une chaîne de caractères et l'ensemble des informations d'état extraites de la portion de chemin retirée se présente sous la même forme qu'une description textuelle utilisée pour déployer un chemin. Ainsi, cette représentation enrichie peut être ultérieurement passée en paramètre à une commande de création (liée ou non à un processus d'insertion) pour réinjecter l'état qui a été capturé au préalable.

Voici maintenant les différentes étapes d'une opération de destruction/reconstruction. On suppose que l'intégralité des instances qui composent un chemin doivent être reconfigurées au niveau de leur code pour faire évoluer la façon dont elles interagissent. Pour chaque type d'extension, le code doit évoluer de la version N à la version N+1 (on suppose que les deux versions de chaque module ont été chargées au sein de l'infrastructure).

1. Une commande de type `RECONSTRUCT` est envoyée à l'instance racine du chemin. Elle contient le numéro de la nouvelle version de code à appliquer à chaque instance.
2. En réaction à cette commande, l'instance racine génère une commande de retrait (qui vise toutes les instances du chemin à l'exception d'elle même) et se l'envoie à elle même.

[15] À cet effet, on pourrait aussi envisager de mettre en place une fonctionnalité de réinstanciation atomique de l'intégralité d'un chemin (comme pour l'infrastructure Click) mais cette approche est plus compliquée à mettre en œuvre que celle que nous proposons, en particulier dans le cas d'une configuration répartie.

3. Le processus de retrait débute par la mise en place d'une barrière de synchronisation au niveau de l'instance racine.
4. Lorsqu'il n'y a plus de commandes en transit au niveau de l'instance racine, le processus de destruction commence. Lors de sa phase d'acquittement, la commande `DESTROY` récupère l'état courant puis déclenche la destruction effective de chaque instance.
5. La commande `DESTROY` revient finalement au niveau de l'instance racine. Avant qu'elle soit détruite, les informations d'état qui ont été extraites sont jointes à la commande `REMOVE` qui l'avait engendrée.
6. La commande `REMOVE` est à son tour acquittée. L'instance racine détecte que celle-ci est associée à une commande `RECONSTRUCT` et récupère la version sérialisée de l'état du chemin. Elle génère ensuite une commande `INSERT` à laquelle elle joint la description du chemin qui a été capturée et les nouveaux numéros de version des modules à utiliser.
7. Le retour de la commande `INSERT` au niveau de l'instance racine signale la fin de l'opération d'insertion terminale.
8. L'instance racine ordonne alors (de façon immédiate) une tâche de mise à jour de son propre code (et demande, via le positionnement d'un fanion, à être avertie de la fin de l'opération).
9. La mise à jour convertit notamment le format de toutes les commandes mises en attente au sein de l'instance racine. Normalement, il n'y a en pas ou peu car la plupart des requêtes en attente sont encore dans la file d'attente du système d'exploitation et n'ont donc pas encore été traduites en commandes internes à Proboscis.
10. La fin de la mise à jour est signalée à l'instance par l'ordonnancement d'un événement.
11. La commande `RECONSTRUCT` est finalement acquittée et les requêtes en attente sont débloquées.

Toutes les étapes de ce protocole sont gérées par l'infrastructure, qui intercepte les commandes au niveau de chaque appel à `prob_cmd_issue` et `prob_cmd_done`. Il n'y a pas besoin d'une prise en charge explicite par les programmeurs d'extensions, y compris pour les extensions racine.

6.4.2.3 Migration d'extensions

Par un procédé relativement proche de celui employé pour la reconstruction d'un chemin, il est également possible de déplacer une ou plusieurs instances au sein d'un chemin. On peut ainsi obtenir les mêmes capacités d'adaptation que celles offertes par un système comme Abacus (cf. 3.2.2.1).

De plus, en combinant les capacités de migration avec les autres mécanismes de reconfiguration (retrait avec capture d'état et réinsertion à un autre endroit), il est possible d'obtenir plus de flexibilité qu'avec les solutions fournies par Abacus.

- Au sein d'Abacus, la supervision du système (collecte de statistiques) et le calcul périodique de la nouvelle configuration optimale sont effectués de manière continuelle par l'infrastructure d'exécution. Avec Proboscis, ces tâches peuvent être confiées à des extensions spécialisées, dont les instances correspondantes peuvent être insérées uniquement aux endroits et aux moments appropriés. Ainsi, la mécanique de supervision est « débrayable » et le coût associé est moindre, en particulier pour les charge d'E/S stables sur d'assez longue périodes.
- Abacus est limité à la migration simple de composants, d'un nœud à un autre, le long d'un chemin réparti. Comme nous l'avons indiqué en 3.2.2.1, cette approche est restrictive car elle ne permet de résoudre que partiellement les problèmes de contention d'accès à une ressource qui impose des contraintes de cohérence (cache, arborescence de répertoires, etc.). Le fait de déplacer tous les composants de gestion de la ressource partagée (un par client) au niveau du serveur

limite le trafic réseau dû aux messages de synchronisation mais n'élimine pas complètement la concurrence au niveau du serveur entre les contextes d'exécution associés aux différentes instances d'un même composant.

Les mécanismes de reconfiguration structurelle offerts par Proboscis permettent, au contraire, de faire fusionner les différents composants déplacés au niveau du serveur. En fonction de l'évolution de la contention sur la ressource partagée, la topologie du système de stockage peut ainsi osciller entre une approche optimiste (gestionnaires au niveau des clients) et une véritable approche centralisée (un seul gestionnaire au niveau du serveur), qui réduit le coût lié à la gestion de la concurrence.

La figure 6.2 illustre une simple migration d'instance (a) ainsi que la fusion de plusieurs gestionnaires au niveau du serveur (b).

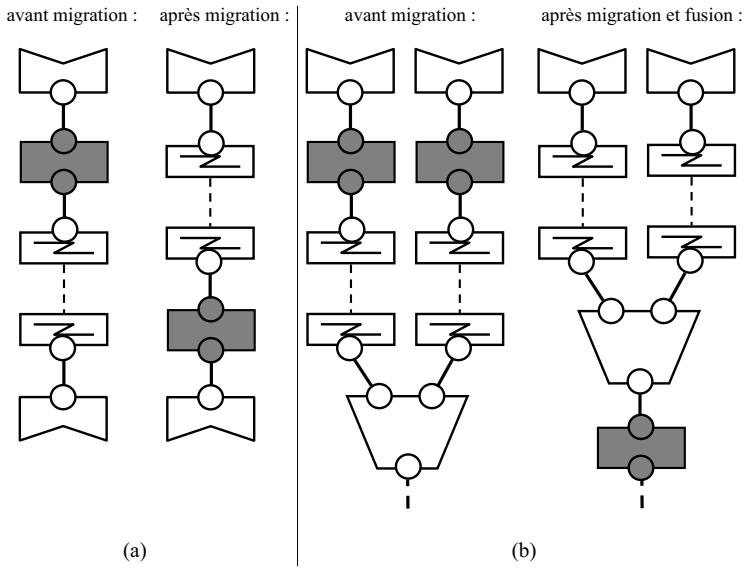


Figure 6.2 – Migration d'une instance au sein d'un chemin

6.5 Protocoles de reconfiguration

Cette section présente deux exemples de protocoles de reconfiguration, qui illustrent comment il est possible de bâtir des opérations de reconfiguration perfectionnées à partir des mécanismes de base décrits précédemment. Le premier concerne un protocole simple permettant de modifier la structure d'un chemin réparti lorsqu'une simple procédure d'insertion/retrait est inadéquate. Le second montre la mise en œuvre d'un protocole de tolérance aux pannes.

Pour simplifier les descriptions, toutes les interactions avec les SEC ne sont pas mentionnées par la suite (en particulier, les acquisitions/libérations de verrous de reconfiguration).

6.5.1 Modification dynamique du réseau employé

Nous considérons ici l'exemple d'un administrateur souhaitant modifier le réseau (physique) utilisé par le système de stockage. Cette opération peut être motivée par :

- la mise à jour du réseau à hautes performances de la grappe (les cartes réseau récentes supportent une mise en service « à chaud » [182]) ;

- le besoin de basculer temporairement le trafic du système de stockage réparti sur le réseau d'administration de la grappe lorsque une intervention de maintenance est nécessaire sur le réseau à haut débit ;
- la nécessité de modifier l'adressage des nœuds en conservant le même réseau physique (à condition qu'une même interface supporte simultanément plusieurs adresses, comme c'est le cas pour Ethernet avec l'adressage IP via le mécanisme d'« alias IP »).

Dans le contexte de Proboscis, ceci implique de modifier la structure des chemins concernés pour remplacer une extension réseau de type R1 par une autre, de type R2. Nous allons maintenant expliquer comment cet objectif peut être atteint.

Remarque : la stratégie de transfert utilisée par chaque extension réseau (spécifiée via un paramètre) n'a aucun impact sur la procédure décrite ci-dessous.

Les principales étapes de la reconfiguration sont illustrées sur la figure 6.3 (le schéma *a* correspond à l'état de départ). Pour simplifier les schémas, le chemin concerné n'est pas représenté dans son intégralité. De plus, les instances A et B peuvent correspondre à n'importe quel type d'extension et être liées à d'autres instances non représentées. Les différentes phases du protocole sont énumérées ci-dessous.

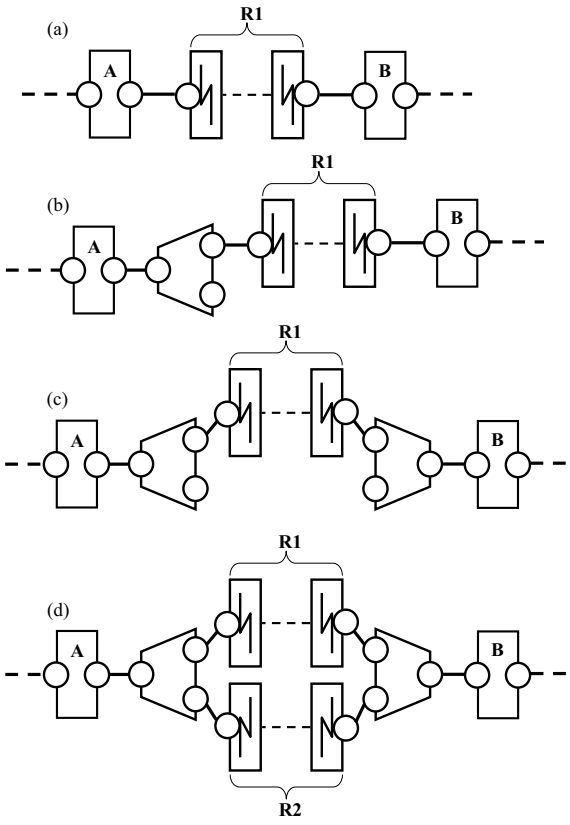


Figure 6.3 – Premières étapes du protocole de reconfiguration dynamique du réseau

1. Suite à un message d'administration émanant du SEC, une commande `NWS` (*network switch*) est envoyée à l'instance A. Les paramètres de la commande incluent notamment la description du nouveau chemin à utiliser pour remplacer la connexion réseau courante.

2. La commande est interceptée au niveau de l'appel à `prob_cmd_issue` par l'instance A. En conséquence, l'infrastructure d'exécution insère successivement deux instances d'interposition :
   - `nws_root`, extension de dispersion, placée en aval de A (schéma *b*) ;
   - `nws_tip`, extension d'agrégation, placée en amont de B (schéma *c*).
3. L'infrastructure d'exécution insère ensuite un nouveau chemin réparti (comme spécifié en paramètre de la commande `NWS`) entre les deux instances d'interposition (schéma *d*).
4. Lorsque la nouvelle connexion réseau est établie, `nws_root` et `nws_tip` basculent leur politique de routage par défaut sur le nouveau chemin (réseau R2). L'ancien chemin (réseau R1) n'est plus utilisé que pour l'acquittement de commandes qui l'avaient emprunté à l'aller.
5. Lorsque qu'il n'existe plus de commandes en transit associées à R1, l'infrastructure retire successivement `nws_root` et `nws_tip`.
6. Finalement, la commande NWS est acquittée.

Ce processus peut être réitéré pour tous les chemins concernés par la migration vers le réseau R2. Puisque le SEC répertorie la configuration courante de tous les chemins déployés au sein de la grappe, il est possible d'automatiser la procédure de reconfiguration à ce niveau en confiant au SEC la responsabilité de déterminer l'ensemble des chemins concernés puis de générer les ordres de reconfiguration adéquats.

Pour finir, il convient d'ajouter que le principe qui vient d'être présenté n'est pas limité à la reconfiguration d'une connexion réseau. Il peut également être employé pour des problèmes d'insertion intermédiaire ou de retrait intermédiaire d'un chemin réparti. Dans ce cas, seul l'emplacement des instances d'interposition varie.

6.5.2 Tolérance aux pannes pour disques dupliqués

Nous considérons à présent une configuration constituée d'un ensemble de clients reliés à plusieurs nœuds de stockage et telle que les données sont intégralement dupliquées (à la RAID 1) sur plusieurs disques. Par souci de simplification, la description ci-dessous est limitée à deux clients et deux disques mais la stratégie employée est généralisable à une topologie quelconque avec n clients et m serveurs.

L'objectif est de pouvoir restaurer le taux de tolérance aux pannes initial suite à la défaillance d'un disque en créant un nouvel exemplaire des données. Les principales phases du protocole sont représentées sur la figure 6.4 (le schéma *a* correspond à l'état de départ). Pour commencer, supposons que le disque n°2 est tombé en panne.

1. La défaillance du disque est détectée par l'instance `mirror` de chaque client, au retour d'une commande d'E/S insatisfaite. En conséquence, chaque instance détruit le chemin qui la relie au disque incriminé, en ayant au préalable contacté le SEC pour obtenir le verrou nécessaire. Ainsi, le serveur est averti de la panne et peut éventuellement relayer l'information auprès d'un administrateur pour suggérer une intervention sur la machine concernée.
2. Les clients entrent, en fait, en compétition pour l'obtention du verrou relatif à l'instance `share` sur le nœud n°2. L'un deux est, par ce biais, implicitement élu maître du processus de reconfiguration (nous supposons par la suite qu'il s'agit du client n°1).

3. Le maître contacte à nouveau le SEC pour lui demander sur quel disque les données doivent être dupliquées. Le SEC maintient une liste de nœuds « de secours » pouvant être utilisée à cet effet. Le choix est effectué en fonction de l'espace de stockage (et éventuellement de critères complémentaires : charge actuelle du nœud considéré, configuration matérielle homogène par rapport aux serveurs restants, etc.). La réponse au client se présente sous la forme d'une description de chemin.
4. Le maître insère alors une instance `copy` en aval de l'instance `share` sur le serveur n°1[16].
5. Le maître déploie ensuite le chemin retourné par le SEC vers le nouveau serveur (n°3), à partir de l'instance `copy`. Ceci aboutit à la configuration du schéma *b* sur la figure 6.4.
6. Le maître envoie une commande `START_COPY` à l'instance `copy`. En réaction, celle-ci commence à cloner le disque n°1 sur le disque n°3. Au cours de cette phase de recopie, les requêtes d'écriture sur disque sont envoyées aux deux disques. Le fait de placer l'instance de recopie juste en amont du disque source permet d'intercepter tout le trafic qui lui est destiné (c'est-à-dire les requêtes d'E/S provenant de l'ensemble des clients) et simplifie ainsi la synchronisation du disque n°3, en cours de clonage.
7. Une fois l'intégralité des données recopiées[17], l'instance `mirror` maître se connecte à l'instance `share` sur le serveur n°3 (cf. figure 6.4, schéma *c*) et envoie une requête d'écriture[18], diffusée notamment sur les deux chemins qui la relient au disque n°3.
8. Lorsque l'instance `share` du serveur n°3 reçoit les deux exemplaires de la requête d'écriture du maître, elle considère que ce dernier est synchronisé avec le nouveau disque.
9. Quand l'instance `copy` est traversée par la requête d'écriture acquittée, celle-ci arrête de transférer les requêtes d'écriture en provenance du maître vers le nouveau disque (car il est désormais directement connecté au serveur n°3). En revanche, il doit continuer à transmettre les requêtes d'écriture originaires du client n°2, qui n'est pas encore synchronisé.
10. À la réception de la requête d'écriture acquittée, l'instance `share` du serveur n°1 remarque que le maître est synchronisé avec le nouveau serveur et avertit les autres clients qu'ils peuvent se connecter au serveur n°3 grâce à la diffusion inversée (cf. 4.3.2.1) d'une commande `UPDATE_MIRROR`[19].
11. Une fois avertie de la présence du nouveau serveur, l'instance `mirror` obtient une description textuelle auprès du SEC et la déploie.
12. Finalement, l'instance `share` du serveur n°1 détecte le moment où tous les clients ont été synchronisés et détruit la connexion entre l'instance `copy` et le serveur n°3, puis l'instance `copy` elle-même.

Contrairement à la plupart des exemples précédents, ce dernier protocole de reconfiguration ne peut être géré de façon transparente vis à vis des types d'extensions concernés. Il est nécessaire que le protocole de reconfiguration soit pris en compte au niveau du code des extensions. Cependant, l'impact des modifications à introduire pour supporter un nouveau protocole de reconfiguration est souvent limité. L'exemple ci-dessus a impliqué la modification de deux extensions existantes (`mirror` et `share`) et le développement d'une extension spécifique au protocole (`copy`).

[16] S'il reste plusieurs exemplaires des données à recopier, le choix du disque source est à la discrétion du maître.
[17] Cet événement est notifié par l'acquittement de la commande `START_COPY`.
[18] La requête est vide (écriture de 0 octets) ; elle sert uniquement de signal de synchronisation.
[19] Il n'y a qu'un seul client concerné (n°2) dans notre exemple.

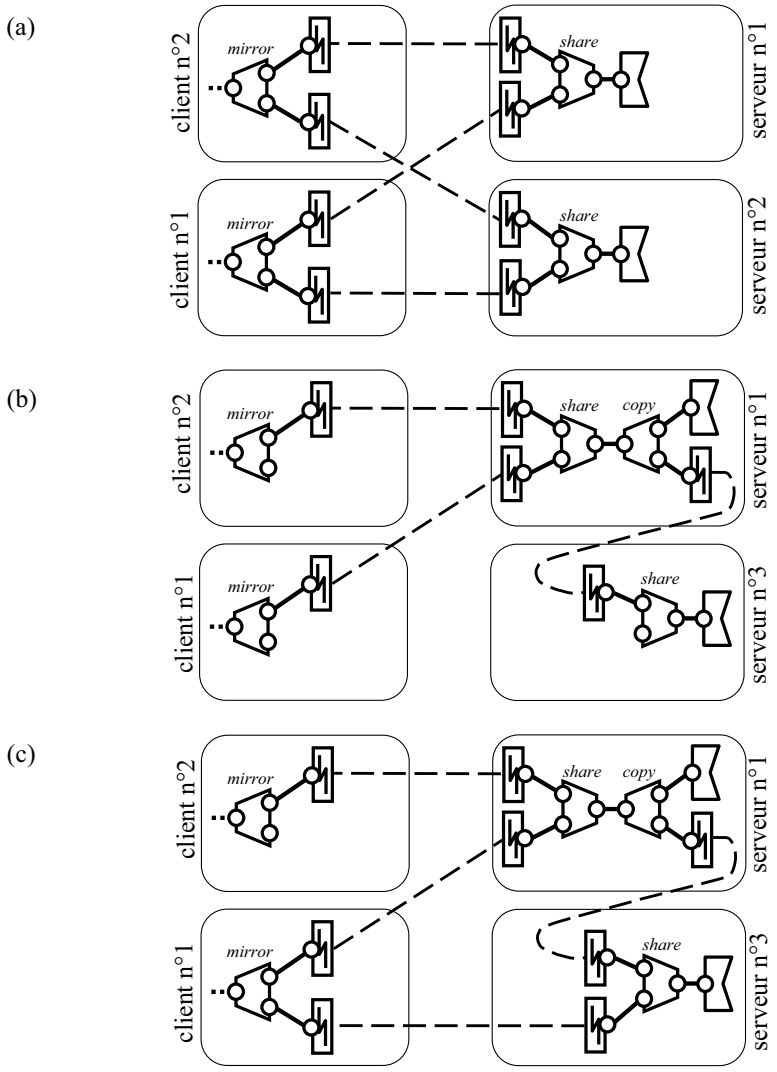


Figure 6.4 – Principales étapes du protocole de reconstruction d'un disque dupliqué

6.5.3 Remarques

Pour terminer ce panorama sur les reconfigurations structurelles, nous pouvons dresser le bilan et apporter quelques précisions sur leur mise en œuvre au sein de Proboscis.

Les mécanismes d'insertion/retrait, de reconstruction de chemin et d'extraction/réinjection d'état sont intégrées à l'infrastructure d'exécution. Ils peuvent être directement utilisés par un administrateur ou servir de base pour le développement d'un protocole plus complexe.

L'introduction d'un nouveau protocole repose sur la définition d'un ou plusieurs nouveaux types de commandes. Dans certains cas favorables, il n'est pas nécessaire de modifier le code des extensions. L'infrastructure d'exécution peut être étendue par l'ajout de traitants de reconfiguration. Ces traitants de reconfiguration sont enregistrés auprès de l'infrastructure d'exécution lors de l'enregistrement du nouveau type de commande auquel ils sont associés. Un traitant donné est appelé par `prob_cmd_issue` ou `prob_cmd_done` lorsque cela est nécessaire. La transparence vis à vis du code des extensions est assurée grâce à ce principe d'interception des appels par l'infrastructure. Ainsi, la

fonctionnalité de modification dynamique du réseau a pu être introduite de manière générique, sans modifier le code des extensions.

L'introduction d'un nouveau protocole nécessite souvent le développement d'une ou plusieurs extensions spécialisées pour l'opération considérée, comme les deux exemples ci-dessus l'ont montré.

Enfin, certaines fonctionnalités, notamment en matière en tolérance aux pannes, imposent une prise en charge du protocole au niveau du code propre aux extensions. Les modifications n'impliquent en général qu'un sous ensemble restreint des types d'extensions utilisées au sein de la configuration considérée.

6.6 Vers un système de stockage autonome

Les mécanismes d'administration et de reconfiguration présentés dans ce chapitre ont pour principale vocation de simplifier le travail de maintenance associé à un système de stockage en grappe. Le serveur central maintient une base d'informations sur la configuration du système et fournit un point central d'administration, qui permet de contrôler l'ensemble des nœuds. La tâche d'un administrateur est ainsi simplifiée mais c'est toujours à lui qu'incombe la responsabilité de surveiller le fonctionnement du système et l'initiative des procédures de reconfiguration.

Par le biais du dernier exemple de protocole présenté en 6.5.2 (restauration du taux de redondance d'une configuration RAID 1), nous avons mis en avant l'idée que les composants du système de stockage peuvent eux-mêmes être à l'initiative d'une opération de reconfiguration. En remplissant la fonction qui lui est assignée, une instance d'extension peut détecter un événement qui motive une reconfiguration (panne d'un disque ou d'un nœud, franchissement d'un seuil de performance critique, etc.) et rapporter l'information au serveur central, voire prendre en charge la procédure de reconfiguration. Dans cette dernière perspective, l'instance doit obligatoirement contacter le SEC, à la fois pour mettre à jour le squelette dont elle dispose (qui lui offre une vue globale de la composition du chemin) et pour obtenir un verrou sur le chemin qu'elle souhaite modifier. Si la demande de verrou échoue, l'instance peut décider de réitérer sa tentative plus tard ou abandonner (en supposant qu'une autre entité, humaine ou logicielle, est déjà en train de mener la reconfiguration adéquate).

Le modèle d'interaction qui vient d'être évoqué nous semble bien adapté pour l'intégration progressive de capacités d'introspection et d'auto-administration au sein du système de stockage, à la fois au niveau des extensions et à celui du serveur central. À terme, le SEC pourrait superviser le fonctionnement du système et, en fonction des indications rapportées par les composants des différents chemins (dont éventuellement des sondes insérées à son initiative), décider des mesures à prendre pour répondre aux pannes et aux problèmes de performances, en déléguant la prise en charge de certains protocoles de reconfiguration aux instances concernées. En ce sens, notre vision rejoint celle du projet « Self-* » (cf. 2.6.2). Pour passer à grande échelle, notre proposition nécessiterait d'ailleurs la même approche de contrôle : partitionnement de la grappe en sous-ensembles, chacun étant administré par un SEC, et fédération arborescente des différents SEC par une hiérarchie de gestionnaires.

Il s'agit clairement d'un objectif à long terme, que nous n'avons pas eu le temps d'approfondir. Nous pensons cependant que ce but est réaliste et que le canevas proposé constitue un substrat intéressant pour développer les fonctionnalités envisagées, pour plusieurs raisons.

- Chaque instance de composant dispose, dans ses méta-données, d'un squelette qui décrit la structure globale du chemin où elle est insérée. Ce mécanisme de réflexivité offre les informations nécessaires à une instance pour adapter son comportement au contexte dans lequel elle

évolue, en particulier lorsqu'une reconfiguration structurelle est nécessaire[20]. Ainsi, un composant peut déterminer s'il est déployé au niveau d'un client, d'un serveur ou d'un éventuel nœud intermédiaire. À partir de cette information, le composant peut par exemple décider de la politique de gestion de ressources à employer ou encore choisir s'il doit prendre l'initiative d'un éventuel protocole de reconfiguration[21].

- Avec notre proposition, les différentes fonctions nécessaires à la mise en œuvre d'un système de stockage de bas niveau (système de gestion de fichier partagé, disque virtuel réparti) peuvent être classées selon trois grandes catégories (extensions d'extrémité, extensions asymétriques et extensions d'interposition), elles-mêmes divisées uniquement en deux sous-catégories. Cette classification simple laisse envisager le développement d'une base de code relativement générique permettant à différents types d'extensions d'analyser le contexte dans lequel ils évoluent (cf. point précédent) et d'adopter un comportement approprié. Par ce biais, une instance de composant, à l'initiative d'un protocole de reconfiguration, peut identifier les autres instances à contacter[22] et décider du mode de routage à associer aux commandes de reconfiguration (il peut être nécessaire d'effectuer une diffusion en présence d'une extension de dispersion, ou une diffusion inversée à partir d'une extension d'agrégation, cf. 4.3.2.1).
- Le SEC possède une vue globale de l'ensemble des chemins déployés sur un groupe de nœuds. Ainsi, si un nœud est traversé par plusieurs chemins distincts, le SEC peut prendre des décisions de (re)configuration de manière conjointe, afin de les faire cohabiter de manière harmonieuse (par exemple, pour éviter la contention sur certaines ressources du système). Autrement dit, l'utilisation d'un gestionnaire centralisé permet d'effectuer des optimisations sur les paramètres du système en recoupant des informations sur certaines de ses parties, dissociées au niveau fonctionnel.
- Le modèle d'interaction développé plus haut permet l'intégration très progressive d'un comportement autonome au sein du système. Il n'y a en effet pas de transition architecturale abrupte entre le statut actuel de l'infrastructure (qui repose sur la supervision d'un administrateur humain) et les améliorations envisagées. Cette évolution pourrait commencer par l'introduction d'une base de règles comportementales au niveau du SEC, pour déclencher des opérations de reconfiguration à gros grain (telles qu'une migration de données). Il conviendrait ensuite de développer, de manière graduelle, les capacités autonomes des différents composants (deux ou trois extensions « intelligentes » par chemin devraient couvrir la plupart des besoins) afin de décharger le SEC et d'aboutir à des procédures plus hiérarchisées. Tout au long de ce cycle de développement, l'architecture du système resterait la même (structuration sous la forme de chemins) alors que les interfaces évolueraient lentement via l'ajout (et l'éventuel retrait) de commandes de supervision et de reconfiguration, associée à une gamme de métriques[23].

De manière plus générale, la perspective de systèmes de stockage véritablement autonomes semble plausible. Le contexte de ces systèmes est en effet associé à des métriques de haut niveau relativement

[20] Ce principe rejoint, par certains aspects, la notion de *flow-based context* développée pour le routeur Click (voir [66], section 2.4) mais cette idée n'est pas employée pour gérer la reconfiguration dynamique.

[21] Typiquement, certaines catégories de reconfigurations pourraient être définies comme étant uniquement à l'initiative des clients (par exemple, le redéploiement de données après une panne de serveur) et d'autres à celle des serveurs (par exemple, la renégociation de garanties de qualité de service ou des stratégies de transfert employées, en fonction de la charge globale du serveur).

[22] en fonction de leur type, ou de leur effet sur les requêtes d'accès aux données en termes de routage et de contenu (cf. 4.2.4.4)

[23] Un défi important à ce sujet concerne la définition d'une interface (jeu de commandes d'administration) assez expressive pour couvrir la majorité des besoins tout en restant relativement restreinte et uniforme pour garantir l'homogénéité du canevas (et ainsi la simplicité de développement et d'assemblage des composants).

claires, du moins en termes d'espace nécessaire, de niveau de redondance des données, de taux de disponibilité et de contrôle d'accès. Les critères de performance sont, quant à eux, moins évidents à spécifier [78].

6.7 Bilan

Ce chapitre a présenté une infrastructure d'administration ainsi qu'un ensemble de mécanismes de reconfiguration dynamique pour les systèmes de stockage en grappe basés sur l'abstraction de « chemin d'E/S » proposée au chapitre 4.

L'infrastructure d'administration repose sur un serveur central, qui maintient un ensemble de méta-données sur la structure et l'état courant du système. Cette topologie permet de simplifier la mise en cohérence des informations d'état et la synchronisation des opérations de reconfiguration et aussi d'offrir un point de contrôle unique à l'administrateur.

Les mécanismes de reconfiguration introduits offrent la possibilité de modifier le code, les paramètres et la structure du système sans interrompre son fonctionnement. Ils ne nécessitent généralement que peu d'efforts de la part des programmeurs de composants. Dans certains cas, une reconfiguration structurelle peut même être gérée de manière totalement transparente par rapport aux composants concernés.

Les fonctions de reconfiguration offertes par l'infrastructure d'exécution peuvent aussi servir de briques de base à des protocoles plus complexes (adaptation de performances par migration de composants, tolérance aux pannes, etc.). Il est ainsi possible de construire des systèmes de stockage plus souples et réactifs.

La plupart des décisions de reconfiguration (et la mise en application qui les suit) sont, au stade actuel de nos travaux, dépendantes d'un administrateur humain. Des exemples développés dans le cadre de protocoles de tolérance aux pannes laissent cependant envisager la possibilité d'injecter progressivement des capacités d'auto-administration au sein du système, réparties entre le serveur central et les composants déployés sur les nœuds impliqués.

Chapitre 7

Evaluation

Sommaire

7.1 Introduction

Cette section commence par décrire notre implémentation prototype de Proboscis puis présente les plates-formes de test et le plan suivi pour la suite du chapitre.

7.1.1 Implémentation prototype

Nous avons pris le parti de focaliser notre implémentation prototype au niveau d'interface bloc. Ce choix est motivé par plusieurs raisons.

- Le développement et la mise au point de code à destination d'un système d'exploitation sont des tâches notoirement complexes et coûteuses en temps. Dans notre cas, la difficulté a été renforcée par le modèle de programmation par événements employé par Proboscis. Il nous a semblé raisonnable de concentrer nos efforts sur un point de validation précis, et *a fortiori*, d'étudier la viabilité de notre approche au niveau le plus bas.
- Disposer d'un système de niveau bloc offre la possibilité d'utiliser différentes types de couches clientes au niveau supérieur : système de gestion de fichiers (ou base de données reposant sur un SGF), base de données travaillant directement sur un périphérique à blocs, partition de swap, etc. En outre, le développement d'un SGF réaliste (en termes de performances et de fonctionnalités telles que la journalisation) aurait probablement été long et quelque peu redondant avec les travaux dans le domaine des SGF empilables.
- Le niveau d'interface à objets apparaît prometteur, notamment par la flexibilité qu'il procure en déportant la responsabilité de l'allocation de l'espace de stockage au niveau des périphériques. Cependant, il n'en existe pour l'instant pas d'implémentation matérielle. Les évaluations de performance sont donc actuellement limitées à des couches d'émulation logicielle, ce qui restreint leur intérêt[1]. En outre, il n'existe que très peu de couches clientes adaptées pour une interface à objets. Expérimenter à ce niveau aurait donc nécessité le développement d'une couche supérieure (traduction fichiers vers blocs) et d'une couche inférieure (traduction objets vers blocs), ce qui rejoint les problèmes mentionnées aux points précédents.

L'évaluation expérimentale de notre modèle n'est donc que partielle à l'heure actuelle mais nous avons cependant veillé, au cours des chapitres précédents, à formuler des propositions compatibles avec les différents niveaux d'interface envisagés. En particulier,

- le chapitre 4 (présentation du canevas) a décrit comment les extrémités d'un chemin Proboscis peuvent interagir avec le système d'exploitation qui les accueille, en fonction du niveau d'interface considéré (4.3.3.3) ;
- les idées développées aux chapitres 5 (gestion des transferts de données) et 6 (reconfiguration dynamique) sont indépendantes du niveau d'interface.

Le prototype de Proboscis a été développé pour le système d'exploitation Linux, et plus particulièrement les noyaux de la série 2.4[2]. Le code est écrit en langage C, selon un modèle orienté objet[3] et correspond à environ 50 000 lignes de code[4] (lignes vides déduites), réparties comme suit :

[1] Sauf, bien sûr, pour prouver la viabilité de l'approche à objets, mais ce point a déjà été traité en abondance, en particulier par le projet NASD de Carnegie Mellon University.

[2] Une partie importante du code est facilement voire directement portable sur la série la plus récente (2.6). Cependant, l'interface de couches d'E/S a été profondément modifiée entre les familles 2.4 et 2.6. En conséquence, les extensions d'extrémité doivent être remaniées de manière significative.

[3] Il aurait été plus naturel d'utiliser un langage conçu pour l'approche objet tel que C++, mais quelques exemples notoires (dont celui de Click) ont montré par le passé qu'il est extrêmement difficile d'intégrer du code C++ au noyau Linux [119].

[4] Cette base de code a été développée avec la contribution de Jørgen Hansen et d'Aurélien Dumez.

- 16000 lignes pour l'infrastructure ;
- 5000 lignes pour les implémentations des différentes versions d'IODSM (soit, en moyenne, 600 lignes par version) ;
- 29000 lignes pour la bibliothèque d'extensions. Une extension de test très simple ne nécessite qu'une cinquantaine de lignes ; les extensions les plus complexes (interaction avec le système d'exploitation) correspondent à 3000 lignes.

L'implémentation actuelle gère les réseaux SCI (via l'API de bas niveau fournie par le pilote IRM et une version modifiée de la couche de communication SciOS [42]) ainsi que n'importe quel périphérique de communication fournissant une interface socket TCP.

7.1.2 Environnement expérimental

La configuration des machines utilisées pour les tests est résumée ci-dessous. Compte-tenu de certaines restrictions (nombre de disques par machine, équipement réseau disponible), il n'a pas été possible de réaliser l'intégralité des expériences avec le même environnement.

Par la suite, le type de machine employé est systématiquement précisé. En l'absence de mention contraire explicite, le système d'exploitation (Linux 2.4.20) est démarré en mode mono-processeur ; les ressources d'exécution supplémentaires (second processeur physique et éventuels processeurs logiques SMT) sont donc désactivées.

Configuration A processeur AMD Athlon 1800 (1533 Mhz) avec 256 ko de cache L2, 1 Go de mémoire DDR 266 Mhz, chipset AMD 760MP, disque dur ATA 133 (IDE) à 7200 tours/minute, contrôleur Gigabit Ethernet Broadcom BCM5701 (sur port PCI 64 bits/66 Mhz), carte SCI Dolphin D330 (sur port PCI 64 bits/66 Mhz), commutateur Gigabit Ethernet HP Procurve 2724, commutateur SCI Dolphin D535 ;

Configuration B 2 processeurs Intel Xeon à 1800 Mhz avec 512 ko de cache L2, 1 Go de mémoire DDR 266 Mhz, chipset Serverworks GCLE, disques dur ATA 133 (IDE) à 7200 tours/minute, contrôleur Gigabit Ethernet intégré Intel 82544GC, commutateur Gigabit Ethernet HP Procurve 2724.

7.1.3 Plan du chapitre

Le reste du chapitre est organisé comme suit.

- Le coût de la modularité du canevas est évalué en 7.2.
- La section 7.3 présente les performances d'accès à un organe de stockage distant en fonction de l'interface de communication et de la stratégie de transfert employées.
- La charge engendrée sur une nœud serveur est examinée en 7.4.
- La section 7.5 étudie le coût et les avantages procurés par la configuration flexible des transferts de données.
- Les protocoles de reconfiguration sont abordés en 7.6
- En 7.7, deux exemples illustrent comment le canevas peut être utilisé pour construire ou améliorer des services de stockage optimisés pour un contexte précis.

7.2 Impact de l'architecture du canevas

Nous considérons ici trois indicateurs du coût de notre canevas : la consommation mémoire associée à son déploiement, l'utilisation du processeur par les composants d'un chemin et l'écart de performances avec les modules fonctionnellement équivalents du système d'exploitation.

Le chargement de l'infrastructure Proboscis et de l'ensemble de la bibliothèque d'extensions nécessite 1,8 Mo de mémoire. La création d'un chemin simple consomme en moyenne 400 ko par nœud traversé, soit environ 100 ko de méta-données par instance de composant.

Nous avons par ailleurs évalué le temps processeur nécessaire pour les fonctions élémentaires qui composent un chemin. Les mesures ont été effectuées sur des machines de type B avec des communications sur Gigabit Ethernet via TCP/IP. Le tableau 7.1 indique le temps d'exécution associé à chaque composant lorsqu'une requête de taille moyenne (64 ko) parcourt un chemin (aller-retour). A titre de référence, la traversée d'un composant « vide » prend environ 2 μs, principalement imputables aux mécanismes d'interception mis en œuvre par les primitives de l'infrastructure `prob_cmd_issue` et `prob_cmd_done`. Au niveau du client, le temps associé à l'extension réseau est décomposé en deux fonctions distinctes : la création/destruction de l'instance globale du bA (ligne *conversion* du tableau) et les communications proprement dites entre le client et le serveur (ligne *transport réseau*). Dans le cas d'une lecture, la majorité du coût lié aux communications au niveau du serveur se situe dans le contexte de l'extension d'interface avec le disque, car les données lues sont immédiatement transmises au client lors de la destruction du BA. Le dialogue avec le disque proprement dit correspond à environ 50 μs, comme on peut le voir pour une écriture.

Nœud	Extension	Lecture	Écriture
Client	racine	21	19
	conversion	14	14
	transport réseau	60	132
	Total	**95**	**165**
Serveur	réseau	10	111
	partage	3	3
	interface disque	107	56
	Total	**120**	**170**

TAB. 7.1 – Utilisation du processeur par les composants d'un chemin pour une requête (durée en microsecondes)

Enfin, nous avons comparé, pour deux configurations centralisées, les performances de deux mises en œuvre différentes, l'une à partir de Proboscis, l'autre basée uniquement sur les services intégrés au noyau Linux. Le premier exemple consiste à comparer un simple PGB Linux à un chemin Proboscis constitué uniquement d'une extension racine et d'une extension terminale, afin d'évaluer le coût de traduction des requêtes d'E/S entre le système d'exploitation et l'infrastructure Proboscis. Le second exemple correspond à un schéma de redondance RAID 1 sur deux disques locaux, réalisé grâce à l'extension `mirror` pour Proboscis et au module `md` pour le noyau Linux.

Dans les deux exemples, la réactivité de la configuration Proboscis est légèrement meilleure (latences des requêtes plus faibles, de 1 à 7%), au prix d'une consommation CPU sensiblement supérieure (sauf pour les grandes tailles de requêtes et les écritures en mode RAID 1). Ces résultats sont

représentés sur les figures 7.1 à 7.4 pour une charge d'E/S séquentielle.

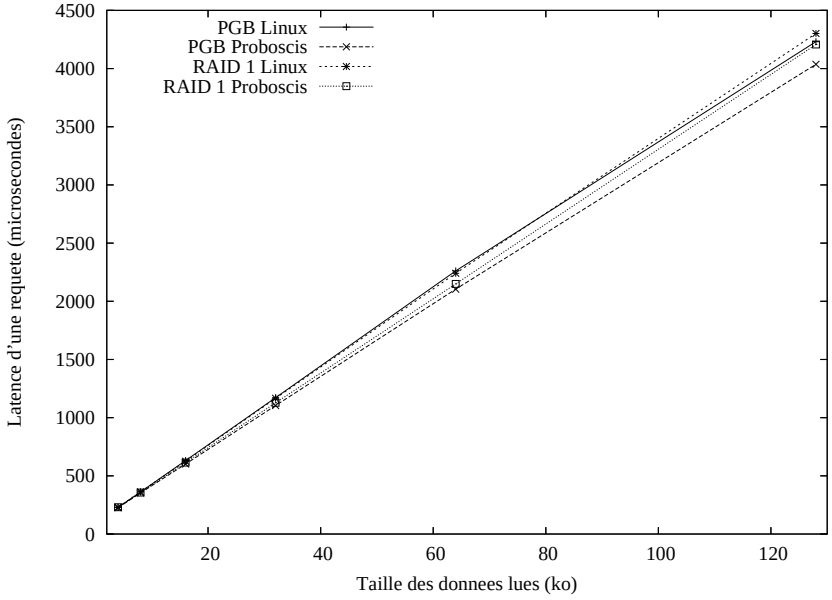


Figure 7.1 – Latence d'une requête de lecture sur pour les configurations locales

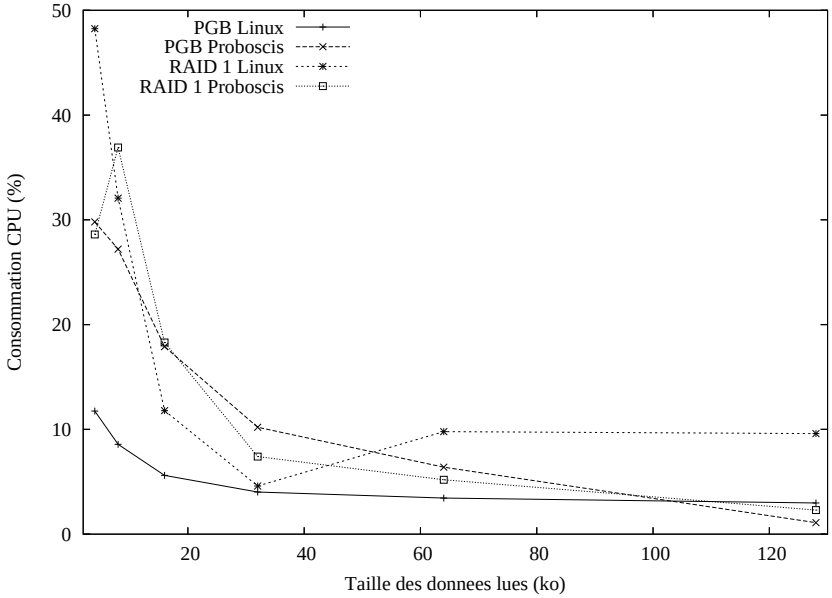


Figure 7.2 – Consommation CPU associée à une requête de lecture sur pour les configurations locales

Ces différentes expériences montrent que les principes retenus pour l'architecture de notre canevas n'imposent pas un surcoût important par rapport aux couches d'E/S « classiques » d'un système d'exploitation et qu'il est ainsi possible d'obtenir des performances comparables, voire meilleures.

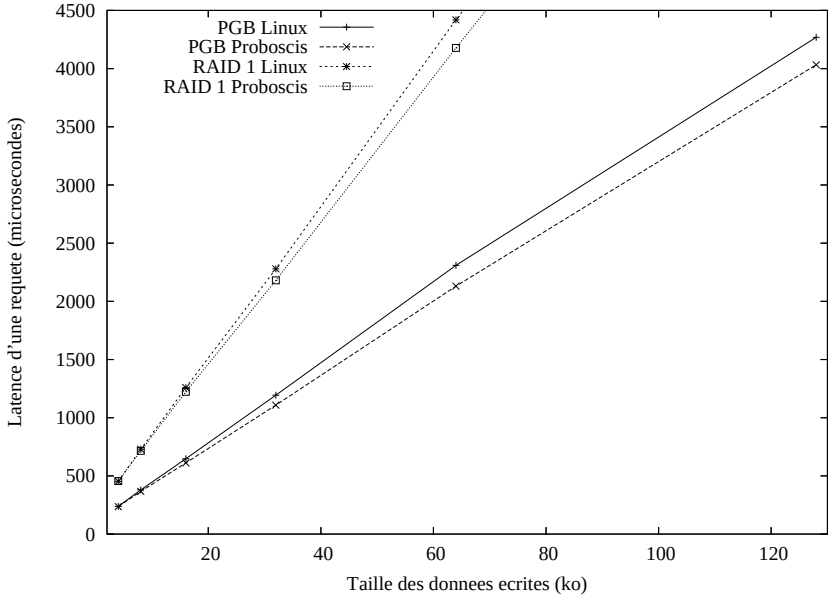


Figure 7.3 – Latence d'une requête d'écriture sur pour les configurations locales

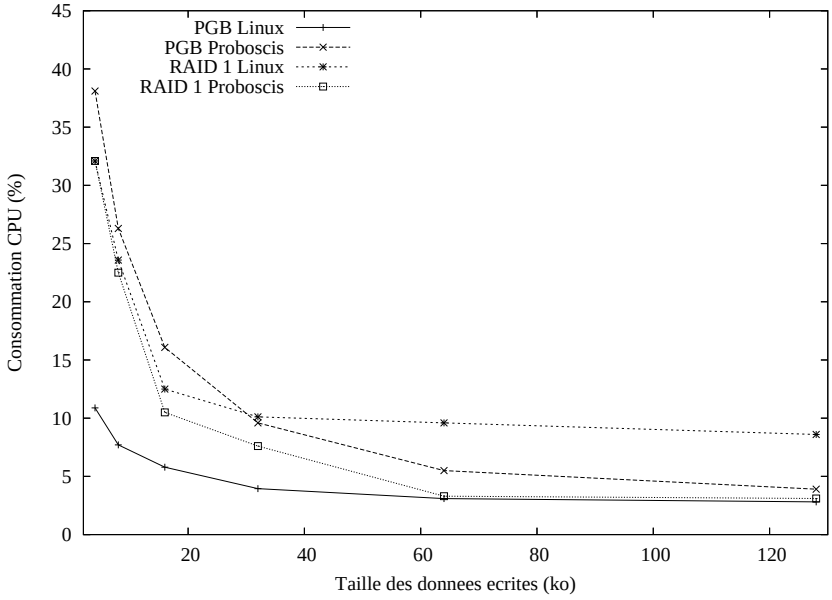


Figure 7.4 – Consommation CPU associée à une requête d'écriture sur pour les configurations locales

7.3 Performance d'accès à un disque distant

7.3.1 Introduction

Cette section résume nos conclusions sur les performances d'accès à un organe de stockage distant. Nous considérons principalement trois dimensions (interdépendantes) : les communications à distance, la technologie d'interconnexion et la stratégie de transfert employée. Par souci de concision, nous ne présentons pas l'ensemble des résultats expérimentaux qui ont été collectés mais seulement des exemples représentatifs des tendances que nous avons observées. Ces exemples correspondent à des mesures de performance élémentaires, facilement reproductibles, obtenues avec des programmes de test tels que Bonnie++ [1] et IOmeter [2].

Avant de décrire les résultats plus détaillés sur les différentes implémentations que nous avons pu observer, nous commençons par résumer les principes généraux déduits de nos comparaisons entre un disque local et un disque contacté via un réseau pour grappe[5]. On peut distinguer trois paramètres principaux pour comparer les performances d'accès à un disque : le débit global offert aux applications, la latence des requêtes, et la consommation CPU associée à la gestion d'une charge d'E/S donnée.

Un disque distant peut fournir la même bande passante globale qu'un disque local (c'est-à-dire traiter la même quantité de données par unité de temps)[6]. Cependant, il n'est pas toujours possible d'égaler la latence d'une requête locale. Ceci est dû à la latence du réseau qui n'est pas toujours négligeable devant celle du disque. C'est le cas pour les accès séquentiels à un disque distant via Gigabit Ethernet. En revanche, les accès aléatoires posent moins de problèmes car la latence de replacement du bras (entre deux requêtes) masque le coût des communications[7]. Toutefois, cet impact sur les performances peut généralement être masqué dans de bonnes proportions au niveau du cache client, en particulier pour les requêtes d'écriture séquentielles. Les opérations les plus désavantagées par cette baisse de performances sont donc les lectures séquentielles. Enfin, la consommation CPU associée à la gestion des E/S est très variable en fonction du protocole employé.

7.3.2 SCI

Les figures 7.5 et 7.6 représentent respectivement la latence et la consommation CPU (au niveau du client) des requêtes de lecture séquentielles sur SCI, en fonction du mode de transfert employé.

La faible latence des couplages à distance permet d'obtenir une réactivité comparable à un disque local à partir de 64 ko. Les transferts par RDMA sont plus pénalisants et imposent un ralentissement d'au moins 12%. Scisocket utilise principalement des couplages pour les petits transferts et le RDMA pour les grandes tailles, en ajoutant un surcoût (1 à 6% de ralentissement supplémentaire) du à son interface de plus haut niveau. La consommation CPU associée aux transferts distants est inférieure à celle d'un disque local, à la fois pour les couplages et le RDMA car l'amorçage des transferts est à la charge du serveur. En revanche, l'utilisation de Scisocket impose des recopies de données qui augmentent fortement (d'un facteur 4 à 6) la charge induite sur le client.

[5] Remarque : La majorité de nos expériences ont été effectuées sur des disques IDE. Néanmoins, des tests sur d'autres machines montrent que ces conclusions sont généralisables à des disques SCSI à hautes performances.

[6] à condition, bien entendu, que le débit du réseau soit suffisant, ce qui est déjà largement le cas aujourd'hui comme nous l'avons vu en 2.1.1.

[7] Cette latence moyenne est aujourd'hui de l'ordre de 9 ms (en lecture) pour les disques de milieu de gamme. Toutefois, la latence des disques du haut de gamme atteint désormais un niveau (5 ms voire moins) tel que les accès aléatoires sont également affectés.

Pour les requêtes d'écriture, les résultats (non représentés) sont similaires aux précédents, en mode *push* comme en mode *pull*.

Une évaluation plus détaillée des performances de Proboscis sur SCI (transferts par couplages et RDMA, sur des machines Pentium II et Pentium III) est proposée dans [90].

La technologie SCI présente plusieurs inconvénients significatifs. D'une part, les performances sont très dépendantes du processeur et du chipset employés. Ainsi, nous avons paradoxalement obtenu de meilleures performances[8], en PIO comme en RDMA, avec des machines plus anciennes mais équipées d'un chipset mieux géré par les cartes SCI. D'autre part, les performances sont très dissymétriques entre les lectures et les écritures (la bande passante des lectures à distance est significativement plus faible), ce qui restreint le choix au niveau des stratégies de transfert envisageables et limite les possibilités en matière d'agrégation de nombreux disques.

Ces problèmes sont propres aux implémentations actuelles du matériel SCI et ne concernent pas (ou du moins pas dans les mêmes proportions) les autres familles de réseaux spécialisés telles que Myrinet ou Infiniband. Ces dernières technologies sont donc, à nos yeux, préférables à SCI pour la mise en œuvre d'un système de stockage à hautes performance sur grappe, d'autant plus qu'elles fournissent des débits sensiblement plus élevés et des prix légèrement plus attractifs.

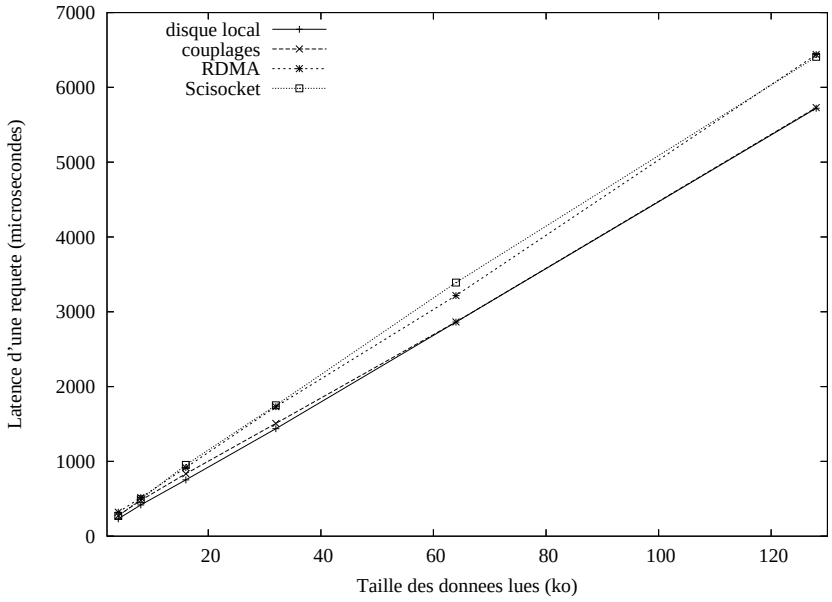


Figure 7.5 – Latence d'une requête de lecture sur SCI

7.3.3 Gigabit Ethernet

Les figures 7.7 et 7.8 comparent les performances en lecture séquentielle d'un disque local avec celles d'un disque distant contacté via Proboscis ou via GNBD (PGB réparti monolithique, cf 2.3.1.1), sur les machines de type B.

Comme nous l'avons indiqué dans l'introduction de cette section, la latence des communications (pile TCP/IP combinée au réseau Ethernet) n'est pas négligeable[9] devant celle de l'accès au disque.

[8] et donc un écart nettement plus faible par rapport à la latence d'un disque local, bien que les disques employés soient des modèles haut de gamme.

[9] La latence des communications s'élève à 77 μs, 600 μs et 1138 mus pour respectivement 4 ko, 64 ko et 128 ko.

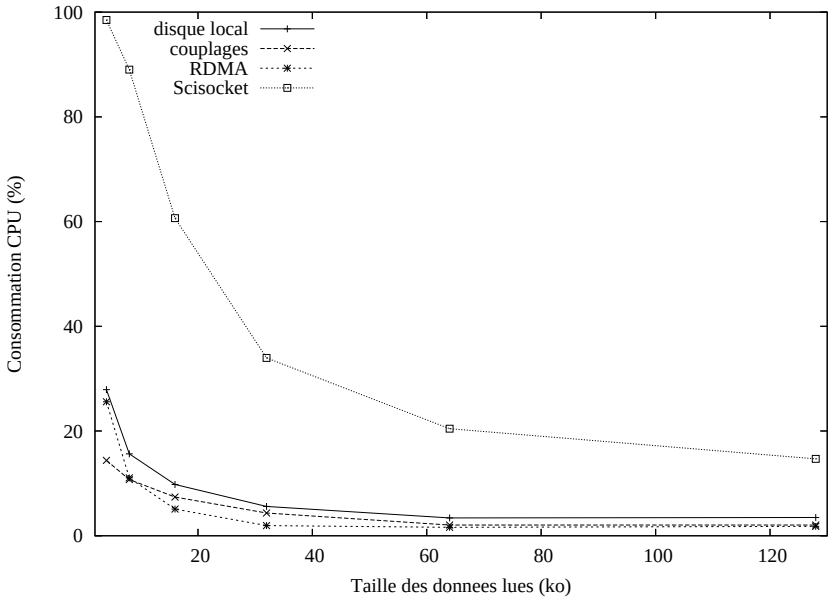


Figure 7.6 – Consommation processeur associée à une requête de lecture sur SCI

Ceci induit un ralentissement significatif des requêtes distantes (plus de 50% pour les petits transferts et environ 25% à partir de 32 ko). Ce phénomène n'est pas lié à l'infrastructure Proboscis mais uniquement aux performances du protocole et du matériel de communication ; GNBD exhibe d'ailleurs exactement la même tendance. On peut toutefois observer un surcoût de 1 à 2% pour les latences de Proboscis par rapport à celles de GNBD. Celui-ci est imputable à l'architecture à composants de notre canevas. L'utilisation de trames Ethernet élargies à 9000 octets (*jumbo frames*) n'améliore que très modérément les latences à partir de 64 ko (d'environ 1 à 2%) et les accroît légèrement pour des tailles inférieures.

Dans tous les cas, la consommation CPU nécessaire à une lecture distante est plus élevée que celle engendrée par un lecture locale : le coût de TCP/IP domine nettement celui de l'armement du DMA d'un disque local et l'écart se creuse lorsque la taille des données augmente. Pour les grandes tailles de données, la consommation est multipliée par 6 pour Proboscis et par 11 pour GNBD. L'utilisation des jumbo frames permet de réduire l'impact de TCP/IP en limitant le facteur d'accroissement entre 1,5 et 2.

On peut enfin remarquer, comme en 7.2, que l'infrastructure Proboscis et le modèle de programmation qui lui est associé n'imposent pas une surconsommation de CPU par rapport aux modules fonctionnellement équivalents du système d'exploitation.

Les écritures donnent des résultats similaires. La latence du modes *pull* est assez proche de celle du mode *push* car la latence d'un message de synchronisation est faible (20 μs).

7.4 Charge sur un nœud serveur

Nous nous intéressons ici à des configurations au sein desquelles chaque nœud joue à la fois le rôle de client et de serveur. Dans ce contexte, nous cherchons à mesurer le coût imposé sur un nœud par le traitement des requêtes en provenance de ses pairs. Nous cherchons également à déterminer l'intérêt de différentes optimisations matérielles pour réduire la charge induite sur le serveur.

Communications Les transferts par RDMA permettent de décharger le(s) processeur(s) du serveur

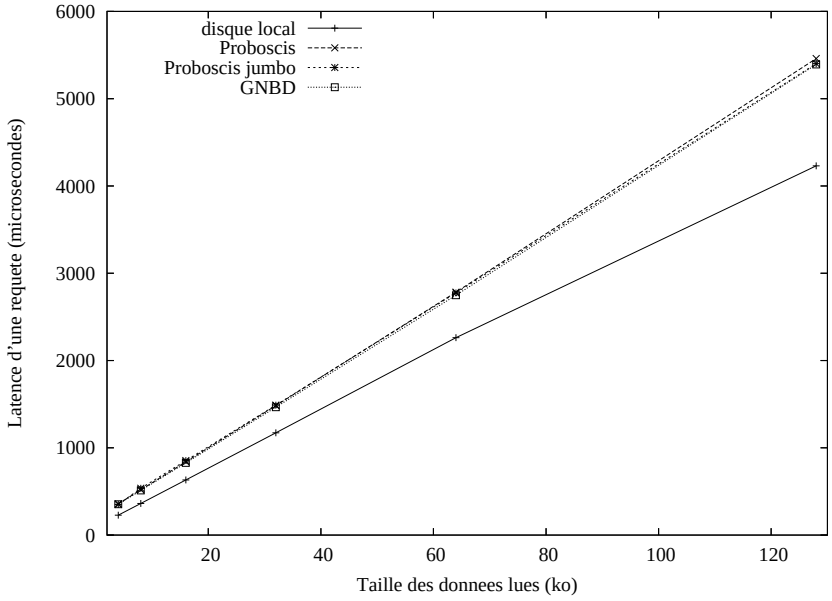


Figure 7.7 – Latence d'une requête de lecture sur Gigabit Ethernet

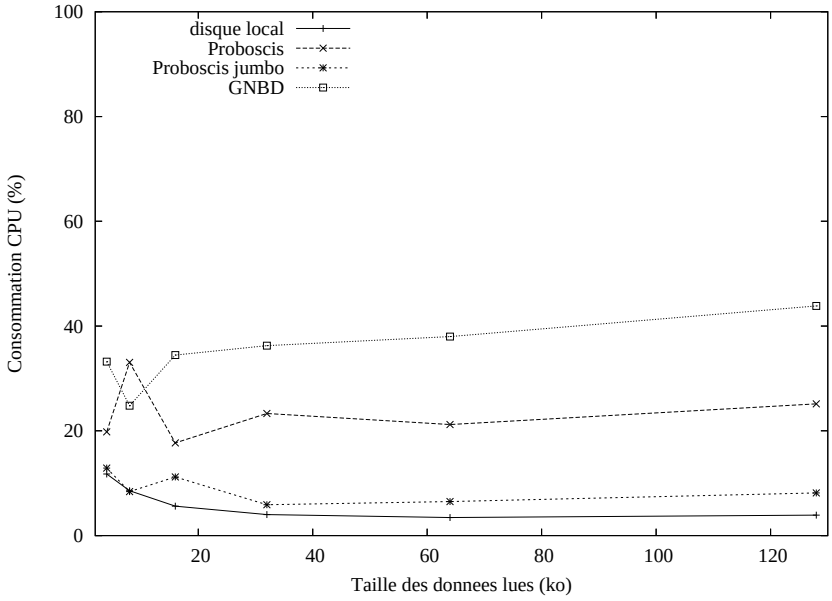


Figure 7.8 – Consommation processeur associée à une requête de lecture sur Gigabit Ethernet

en évitant les traitements associés à une pile protocolaire telle que TCP/IP (qui incluent notamment des recopies de données) ainsi qu'en réduisant le nombre d'interruptions. Pour Gigabit Ethernet, l'utilisation de trames plus grandes (*jumbo frames*) contribue à réduire l'impact de la pile réseau et le nombre d'interruptions au niveau du serveur.

Support d'exécution Comme nous l'avons expliqué en 1.2.1, les architectures de processeurs sont en train d'intégrer davantage de support pour parallélisme d'exécution. Nous cherchons à estimer dans quelle mesure ces capacités de traitement accrues peuvent contribuer à « absorber » le coût des requêtes distantes d'accès aux données.

Notre évaluation repose sur l'observation du ralentissement imposé à une application (séquentielle) s'exécutant sur le serveur en présence d'une forte charge d'E/S issue d'un client distant. La charge d'E/S injectée correspond à un flot intensif de requêtes séquentielles en lecture ou écriture. Au niveau du serveur, deux applications ont été utilisées. La première, `calc`, est une simple boucle de calcul qui n'utilise que quelques variables. Elle permet de mesurer l'impact des interruptions et des changements de contexte dus aux requêtes de stockage. La seconde application est une version séquentielle de `bt` (extrait du *NAS Parallel Benchmark*, classe A) qui effectue des calculs en virgule flottante et de nombreux accès à la mémoire (300 Mo lui sont alloués) mais pas d'E/S. Ce second test permet d'estimer l'impact de la contention sur le bus mémoire et de la pollution des caches au niveau du (des) processeur(s).

Le surcoût représenté sur les figures correspond à l'augmentation du temps d'exécution de l'application sur le serveur en présence d'E/S par rapport à une exécution « à vide », avec la même configuration. Pour toutes les expériences, aucune variation sensible (supérieure à 1%) n'a été détectée au niveau des performances d'accès perçues par le client injecteur de charge par rapport à une situation où le serveur n'exécute pas d'application.

Influence du parallélisme matériel Les figures 7.9 et 7.10 présentent les résultats obtenus sur les machines de type B, en fonction du test exécuté sur le serveur (`calc` ou `bt`) et du profil de la charge d'E/S injectée (lecture ou écriture). Le trafic de stockage est véhiculé sur Gigabit Ethernet, via l'extension réseau TCP/IP de Proboscis et les transferts de données sont effectués à l'initiative du serveur.

Pour chaque cas, différentes configurations ont été utilisées :

- *mono* : un seul processeur physique est activé, le mode SMT est désactivé ;
- *SMP* : deux processeurs physiques sont activés, le mode SMT est désactivé ;
- *SMT* : un seul processeur physique est activé, le mode SMT est activé (deux processeurs logiques sont donc visibles par le système d'exploitation) ;
- *MPT* : deux processeurs physiques sont activés, le mode SMT est activé (quatre processeurs logiques sont donc visibles par le système d'exploitation).

On peut tout d'abord remarquer que le surcoût imposé par les requêtes d'écriture est nettement supérieur à celui lié aux requêtes de lecture. Ceci s'explique principalement par le fait que le serveur doit gérer une communication supplémentaire avec le client et surtout par les interruptions induites par la réception des données.

Dans le cas de `calc`, on voit que l'utilisation d'un second processeur permet d'absorber une partie importante des traitements liés au service de stockage. L'apport du mode SMT est également sensible mais moindre. Enfin la combinaison SMP-SMT masque presque totalement les coûts au niveau de l'application.

Pour une application plus réaliste comme `bt`, l'impact du service de stockage demeure sensible, même dans le cas le plus favorable (environ 10% en lecture et 20% en écriture). Cependant, l'ajout de ressources d'exécution restreint fortement le ralentissement (division par deux, voire trois, par rapport au pire résultat). Ce constat s'applique notamment au mode SMT, qui permet d'obtenir un bon niveau de parallélisme d'instructions car `bt` et Proboscis n'utilisent généralement pas les mêmes unités de traitement au sein du processeur (`bt` utilise majoritairement l'unité de calcul flottant).

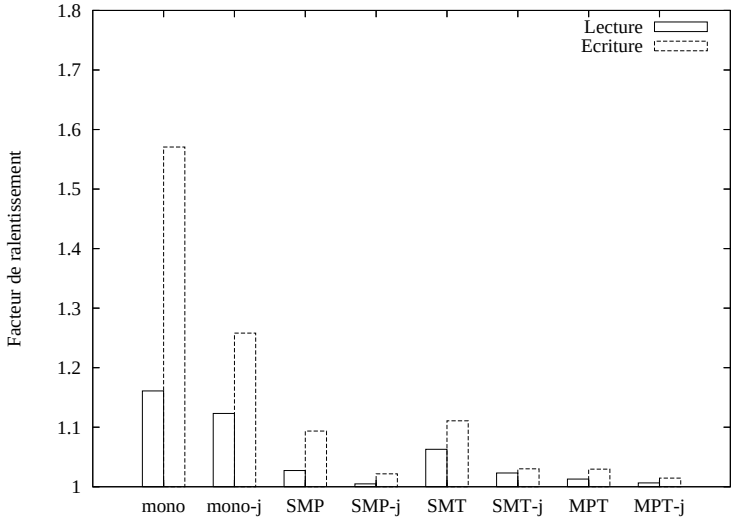


Figure 7.9 – Ralentissement de l'application `calc` sur un nœud serveur par la charge d'E/S avec Gigabit Ethernet

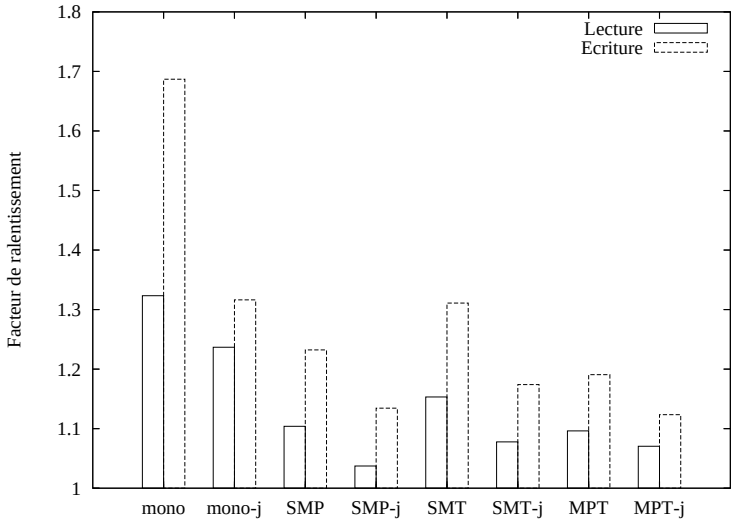


Figure 7.10 – Ralentissement de l'application `bt` sur un nœud serveur par la charge d'E/S avec Gigabit Ethernet

Influence des jumbo frames Les figures mentionnées au paragraphe précédent incluent également les résultats obtenus en activant le mode *jumbo frames* des cartes Gigabit Ethernet[10], avec la taille de trame maximum (9000 octets)[11].

L'impact des grandes trames Ethernet est perceptible, en particulier pour les requêtes d'écriture. Ce mécanisme permet de mieux amortir le coût du protocole Ethernet et en particulier de limiter le nombre d'interruptions sur le serveur.

En combinant l'ajout de ressources d'exécution avec l'utilisation de grandes trames, il est possible d'absorber presque intégralement le surcoût imposé à `calc` et de limiter celui-ci aux alentours de 10% pour `bt`.

Influence d'un réseau spécialisé Les expériences ont été répétées sur les machines de type A afin d'évaluer l'apport d'un réseau spécialisé tel que SCI. Les résultats sont présentés sur les figures 7.11 et 7.12 pour les stratégies de transfert basées sur les couplages à distance, le mécanisme de RDMA et l'utilisation de l'interface socket. Le coût des écritures est, là encore, supérieur en raison des interruptions supplémentaires. On peut constater que dans le cas des couplages directs et du RDMA, le surcoût imposé au serveur est relativement faible et varie peu entre `calc` et `bt` (environ 5% et 8% respectivement).

En revanche, les transferts basés sur Scisocket chargent nettement plus le serveur. Ceci illustre le fait qu'une couche de communication offrant une interface socket, même en court-circuitant la pile protocolaire TCP/IP, ne rivalise pas forcément avec une interface de plus bas niveau en ce qui concerne la consommation de ressources. Scisocket est avant tout optimisée pour fournir une très faible latence pour de petits messages et n'est pas destinée, en priorité, à offrir un bon compromis entre performances et coût pour la gestion de gros échanges de données.

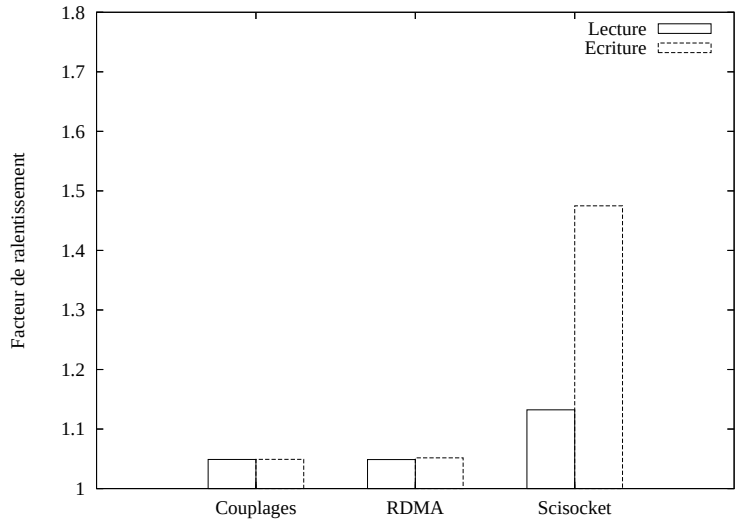


Figure 7.11 – Ralentissement de l'application `calc` sur un nœud serveur par la charge d'E/S avec SCI

Impact sur les communications Nous avons également cherché à évaluer l'impact du trafic réseau lié au stockage sur les communications applicatives du serveur. Pour SCI comme pour Gigabit

[10] Ces résultats sont repérables par leur suffixe « -j ».

[11] Ne disposant pas d'un commutateur Gigabit Ethernet administrable, les mesures ont été effectuées en reliant directement un client au serveur via un câble croisé.

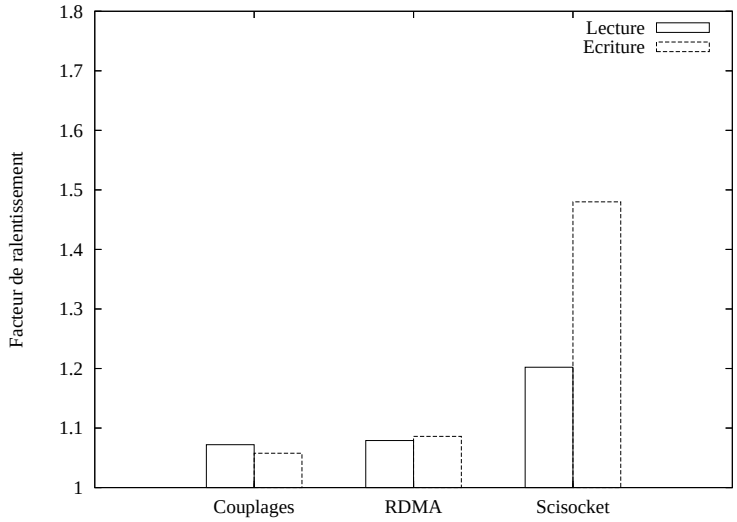


Figure 7.12 – Ralentissement de l'application `bt` sur un nœud serveur par la charge d'E/S avec SCI

Ethernet, la réduction de bande passante est équivalente au débit du disque. En d'autres termes, les protocoles de communication employés pour le stockage réparti ne « gaspillent » pas une part sensible de la bande passante disponible.

La lourde charge de requêtes associées au stockage cause cependant une augmentation de latence pour les communications du serveur. Celle-ci s'élève en moyenne à 11% pour les communications par couplages directs SCI et à 23% pour les transferts par RDMA. Pour les communications par sockets TCP, la latence est augmentée de 5% sur GigabitEthernet et de 14% avec Scisocket.

Ces expériences montrent que l'utilisation de chaque nœud d'une grappe à la fois comme ressource de calcul/traitement et comme brique de stockage partagée est une approche viable. De plus, nous avons pu observer que les différentes optimisations intégrées au matériel peuvent contribuer à réduire substantiellement la dégradation de performances imposée aux applications s'exécutant sur le serveur.

Il est intéressant de remarquer qu'une part importante de ces optimisations sont (ou vont être) disponibles sur une grande majorité de machines et constituent un axe prioritaire de développement pour les constructeurs. C'est notamment le cas pour les microprocesseurs, dont le support pour le parallélisme et la taille des caches vont croître. Les équipements réseau spécialisés nécessitent encore des investissements non négligeables mais les prix tendent à diminuer, qu'ils s'agisse des commutateurs Gigabit Ethernet administrables (permettant l'emploi de *jumbo frames*) ou des interfaces de communication capables de RDMA.

Enfin, l'évaluation que nous venons de présenter ne constitue qu'une première étape et n'a pas considéré l'influence de tous les paramètres. En particulier, deux pistes nous semblent intéressantes. D'une part, étudier l'impact du service de stockage sur les performances d'une application parallèle (à la fois au niveau de la grappe et à l'échelle d'un nœud). D'autre part, en ajoutant des ressources matérielles consacrées au stockage (disques et interfaces réseau), déterminer jusqu'à quel point un nœud peut jouer convenablement le double rôle d'exécution d'applications et de serveur de stockage.

7.5 Gestion configurable des transferts de données

Cette section est organisée en deux parties. La première évalue le coût induit par la nouvelle approche de gestion des transferts que nous avons proposée et la seconde justifie l'intérêt d'un découplage potentiel entre les messages de contrôle et les transferts de données.

7.5.1 Impact d'une infrastructure de transfert flexible

L'objectif est ici d'évaluer le surcoût associé à notre proposition par rapport à une approche monolithique. La diminution de performances que nous cherchons à quantifier est causée par deux facteurs : d'une part, le coût des liens d'indirection de l'infrastructure IODSM et surtout, celui imposé par le modèle de programmation basé sur des assemblages de tampons répartis.

Dans cette optique, nous avons comparé les performances de deux implémentations différentes d'une même stratégie de transfert, l'une étant basée sur notre modèle de programmation et l'autre correspondant à une version plus « traditionnelle » (mais sans aucun levier en termes de flexibilité). Ces deux implémentations seront, par la suite, désignées respectivement par *version IODSM* et *version simple*. Plus précisément, nous avons effectué cette comparaison en choisissant une stratégie représentative en fonction de l'API de communication considérée :

- pour TCP/IP, les transferts de données sont couplés aux messages applicatifs ;
- pour SCI, les transferts sont effectués par RDMA et sont déclenchés à l'initiative du serveur, comme dans le cas du *Direct Access File System* (DAFS, cf. 2.4.2.4).

L'évaluation a été effectuée dans le contexte de Proboscis, selon deux critères : la complexité de programmation et les performances des transferts.

Complexité du code Il est difficile de comparer objectivement deux implémentations d'une même stratégie, d'autant plus que la complexité du modèle de programmation par événements adoptée par Proboscis prédomine, et tend ainsi à niveler les difficultés liées à la mise en œuvre des transferts de données. D'une manière générale, une implémentation basée sur notre modèle de programmation est plus verbeuse mais permet de distinguer clairement le code de l'application de celui correspondant aux transferts de données.

Le nombre de lignes de code nécessaires pour implémenter les différentes extensions qui composent un chemin réparti donne une idée du travail supplémentaire demandé par notre approche. Pour TCP/IP, 2735 lignes sont nécessaires pour la version simple (extensions) contre 3680 pour la version IODSM (extensions et module IODSM), soit 34% d'augmentation. Pour SCI, le surcoût est moindre : 7% (avec respectivement 3480 et 3730 lignes). Ces estimations sont pessimistes car elles incluent, pour une interface de communication donnée, une portion significative de code (environ 500 lignes) mutualisée pour différents IAS.

Performance des transferts Différents bancs d'essais (*benchmarks*) ont été employés pour caractériser le surcôut de notre modèle de programmation. Certains tests ont été effectués au niveau blocs (divers profils de requêtes avec IOmeter et dd), d'autres au niveau fichiers avec le SGF Ext3 et un cache de blocs initialement vide (copie de fichiers et de répertoires, compilation d'un noyau Linux, Bonnie++, Iozone [3], etc.).

Les tests menés avec un « véritable » disque dur au niveau du serveur ne mettent pas en évidence une différence de performances significative entre les deux implémentations d'une même stratégie : la grande latence des accès (physiques) au disque masque le retard imposé par l'infrastructure IODSM. Pour tirer des conclusions plus précises et plus générales, les expériences ont été répétées avec un

ramdisk. Même dans cette configuration, la majorité des tests continuent à fournir des résultats équivalents avec les deux implémentations. Des écarts de performances sont toutefois perceptibles dans le cas particulier où les tampons de données sont de petite taille et où les requêtes sont envoyées de manière purement séquentielle (sans aucun parallélisme). Ce phénomène est observable sur la figure 7.13 qui représente l'écart de latence entre les deux implémentations pour une taille de tampon donnée. On peut cependant constater que le ralentissement maximum est relativement faible (7%), et que les gros transferts (64 ko ou plus) ne sont pas concernés par ce problème (la latence supplémentaire introduite par le modèle de programmation est alors négligeable devant celle du transfert).

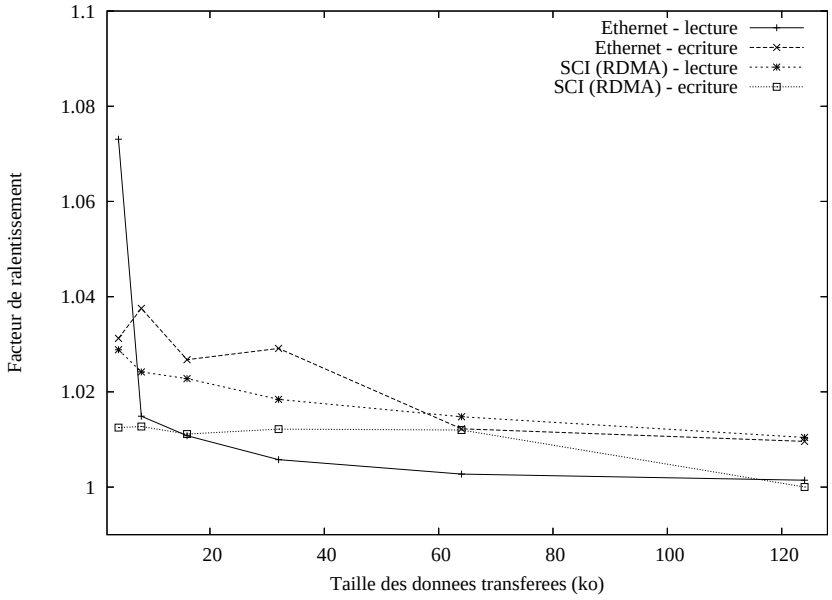


Figure 7.13 – Ralentissement des transferts de données du à l'infrastructure IODSM

7.5.2 Découplage des messages de contrôle et des transferts de données

Nous illustrons à présent l'intérêt d'un découplage entre messages de contrôle et transferts de données dans certaines situations par le biais de deux exemples.

Allocation retardée de tampons Considérons le cas d'un serveur gérant plusieurs classes de clients, associées à différentes priorités. Si nécessaire, les requêtes de faible priorité sont mises en attente au niveau du serveur.

Dans le cas où les clients émettent en majorité des requêtes d'écriture (auxquelles sont joints les tampons de données correspondants), cela peut aboutir à une situation défavorable où une grande part des ressources du serveur est paradoxalement utilisée pour la gestion des requêtes non prioritaires, ce qui tend à accélérer la saturation du serveur. Par opposition, le découplage des requêtes et des transferts de données contribue à un meilleur passage à l'échelle du serveur, en lui permettant de repousser l'allocation des tampons de données jusqu'au moment de leur traitement effectif.

Nous avons implémenté ce scénario avec deux classes de clients, chacun d'entre eux émettant une rafale de requêtes d'écriture (comprises entre 32 et 128 ko). Les expériences ont été effectuées sur les machines de type A, sur le réseau Gigabit Ethernet.

Avec quatre clients, dont trois de faible priorité, on peut observer une diminution nette de la mémoire occupée (de 778 Mo à 335 Mo, soit un gain de 56%) avec, en contrepartie, une augmentation

modérée de la latence des requêtes (9%)[12]. Avec sept clients, la première stratégie (données jointes aux requêtes) écroule le serveur, alors que la seconde méthode de transfert (découplage) lui permet de disposer d'assez de mémoire libre (130 Mo) pour continuer à assurer sa fonction.

Suppression des recopies de tampons sur les nœuds routeurs Nous nous intéressons maintenant à une topologie au sein de laquelle un nœud routeur sert d'aiguilleur entre les clients et un ensemble de serveurs de stockage. Si les données manipulées sont jointes aux requêtes de lecture/écriture, celles-ci sont inutilement recopiées au niveau du routeur (qui n'y accède pas). En conséquence, des ressources du routeur sont gaspillées et la latence globale des requêtes est accrue (même lorsque le système est faiblement chargé).

Les expériences ont été menées sur les machines de type A, avec le réseau Gigabit Ethernet, avec différentes applications de test. Même avec une faible charge d'E/S (et donc pas de contention au niveau du nœud routeur), la latence des E/S est réduite de façon sensible par des transferts de données directs entre clients et serveurs : en moyenne, de 4 à 9% pour les petites tailles (4 et 8 ko) et de 30 à 43% à partir de 64 Ko.

7.6 Reconfiguration dynamique

Cette section s'intéresse aux performances des mécanismes de reconfiguration dynamique présentés au chapitre 6. Elle commence par décrire les performances des mécanismes élémentaires d'administration. Elle détaille ensuite l'évaluation des deux protocole de reconfiguration décrits en 6.5 et basés sur des modifications de la structure d'un chemin.

7.6.1 Mécanismes de base

Le déploiement d'un nouveau module (ou d'une nouvelle version d'un module déjà déployé) sur un nœud consiste à transférer un fichier binaire (50 à 200 ko) via le réseau d'administration, à charger le module au sein du noyau et à enregistrer l'extension auprès de l'infrastructure Proboscis. Cette procédure n'interfère pas avec le fonctionnement du système de stockage (pas de commandes bloquées) et nécessite environ 40 ms (principalement inputables au chargement du module par le noyau).

La modification d'un paramètre d'un composant implique l'envoi d'un message d'administration sur le nœud où débute le chemin visé, l'envoi d'une commande `UPDATE_ATTR` le long du chemin et l'exécution du traitant de reconfiguration au niveau de l'instance concernée. Dans la plupart des cas, la reconfiguration du paramètre consiste simplement à modifier une variable. Le coût de la reconfiguration est alors dominé par les communications et l'ensemble de l'opération nécessite environ 2 ms. Cependant, dans certaines situations moins fréquentes, le traitant de reconfiguration est plus complexe (par exemple, le redimensionnement d'un cache ou d'une table interne peut entraîner des allocations/libérations/recopies de zones mémoire). La durée nécessaire pour la reconfiguration est alors assimilable au temps d'exécution du traitant.

Le temps de mise à jour du code d'une extension est dominé par le transfert d'état entre l'ancienne et la nouvelle instance. Le passage par une représentation canonique de l'état sous forme textuelle est coûteux et peut prendre plusieurs dizaines de millisecondes. Ceci plaide en faveur de l'introduction

[12]Cette implémentation n'est pas totalement réaliste car elle court-circuite les mécanismes (rudimentaires) de contrôle de flux au niveau du client (l'ordonnanceur de disque n'autorise qu'un nombre limité de requêtes en cours). Ainsi, un tel niveau de charge sur le serveur ne peut, en général, pas être atteint avec le faible nombre de clients déployés. Notre objectif principal était cependant de comparer la consommation de ressources induite par une charge importante sur le serveur avec les deux approches.

d'un protocole de négociation (cf. 6.2.2), qui permettrait, dans la plupart des cas, un transfert direct de l'état entre l'ancienne et la nouvelle version du composant.

7.6.2 Protocoles de reconfiguration

7.6.2.1 Modification du réseau

Nous avons implémenté le protocole de modification dynamique du réseau présenté en 6.5.1. Dans notre exemple, un chemin repose initialement sur une extension réseau spécialisée pour SCI, remplacée par une extension réseau pour TCP/IP (associée à une interface Gigabit Ethernet).

Pour commencer, nous avons mesuré la durée nécessaire pour chaque étape du protocole, sans trafic au sein du chemin. Les opérations du côté serveur (telles que l'insertion de `nws_tip` ou la suppression de l'ancien chemin) prennent en moyenne 340 μs contre 35 μs du côté client (par exemple, l'insertion de `nws_root`). Vu de la console d'administration, l'ensemble du protocole s'exécute en 2,4 secondes (pour un seul chemin, mais de multiples chemins peuvent être reconfigurés en parallèle).

Nous avons également observé l'exécution du protocole de reconfiguration en concurrence avec un flux intensif de requêtes d'écriture (128 ko) au sein du chemin. Le protocole de reconfiguration n'est pas ralenti par le flux de requêtes.

En revanche, le trafic du système de stockage est perturbé par les opérations d'administration. Déterminer l'impact de la reconfiguration sur le transfert de données n'est pas évident car les performances des deux réseaux sont nettement différentes. Pour cela, nous adoptons une méthodologie particulière. Les performances du transfert sont d'abord mesurées sur chaque réseau sans reconfiguration concurrente. Ensuite, le test est répété en déclenchant l'opération de reconfiguration au bout d'un temps précis. On connaît ainsi le temps d'utilisation de chaque réseau et il est alors possible de déterminer le ralentissement imposé par la reconfiguration.

En moyenne, 260 requêtes sont retardées à cause des barrières de synchronisation mises en place par les commandes de reconfiguration structurelle. La latence de ces requêtes subit une augmentation moyenne de 30% mais ne dépasse pas la valeur maximum observée dans des conditions normales (sans reconfiguration). À titre indicatif, dans le cas d'un transfert de 1 Go, la reconfiguration impose un ralentissement de 11% au niveau du débit global et 3% des requêtes sont retardées.

7.6.2.2 Tolérance aux pannes pour disques dupliqués

Cet expérience correspond au protocole décrit en 6.5.2, avec deux clients et deux serveurs reliés par Gigabit Ethernet (machines de type A).

Les temps nécessaires pour les opérations élémentaires de reconfiguration sont comparables à ceux obtenus en 7.6.2.1. Sans charge d'E/S concurrente, l'extension `copy` peut cloner les données sur le nouveau serveur à la vitesse du disque (en consommant environ 40% du CPU sur le serveur « source » et 60% sur le serveur « destination »).

Pour évaluer les performances de la cohabitation entre les requêtes d'E/S émises par les clients et le processus de synchronisation du nouveau disque, des flux intensifs de requêtes ont également été lancés au niveau des clients. Afin de simuler une situation peu favorable, aucun cache n'est employé au niveau des serveurs et chaque client travaille sur une zone différente du disque. L'implémentation actuelle de l'extension `copy` est relativement simple : elle recopie les données de manière séquentielle sur le nouveau disque et lui retransmet également toutes les requêtes d'écriture. La cadence des recopies est contrôlée par un boucle de régulation, en fonction de l'évolution de la latence des requêtes. En jouant sur les paramètres de contrôle de la boucle, différents compromis peuvent être atteints. Du point de vue des clients, la diminution des débits peut être limitée à 11%, en imposant

un ralentissement de 52% au processus de clonage. Inversement, ce coût peut être limité à 6%, en imposant une pénalité de 45% aux E/S des clients. Les effets de la concurrence entre les accès des clients et le processus de reconstruction pourraient probablement être amoindris grâce à des optimisations plus fines (gestion d'un cache pour les clients au niveau du serveur, meilleur ordonnancement des requêtes de lectures associées au clonage, etc.).

Dans le cas où un seul client est connecté au système de stockage, on peut également comparer la méthode de reconstruction employée par notre protocole (transfert de données direct entre le serveur restant et celui en voie de reconstruction) et la méthode plus classique où les données à copier transitent par le client. Sans charge d'E/S concurrente, la synchronisation directe entre les deux serveurs permet un gain en temps de 31%.

7.7 Développement de services de stockage optimisés

Nous illustrons cet aspect en revisitant successivement deux techniques largement étudiées lors des quinze dernières années : la pagination à distance et les caches coopératifs.

7.7.1 Pagination à distance

L'idée de base est simple : utiliser des ressources d'autres machines (mémoire, disques) pour améliorer les performances de pagination d'un nœud dont l'espace de travail (*working set*) est nettement supérieur à sa capacité mémoire[13]. Nous décrivons ci-dessous trois exemples (tirés de la littérature) représentatifs des infrastructures logicielles développées à cet effet.

- Le système GMS (*Global Memory Service*) [69] est intégré finement aux mécanismes de mémoire virtuelle d'un système d'exploitation. Comme son nom l'indique, il permet une gestion globale de la mémoire des nœuds d'une grappe. En particulier, un nœud (X) peut utiliser une partie de sa mémoire comme cache de pages d'un autre nœud (Y). Seules des pages « propres » (c'est-à-dire déjà sauvegardées sur une partition de swap locale à Y) sont mises en cache. Ainsi, le nœud X peut, sans préavis, réduire la taille du cache offert à Y lorsqu'il a besoin de davantage de mémoire pour son propre usage.
- Un autre système pionnier, RMP (Reliable Memory Pager) [142], a été basé sur un périphérique à blocs au niveau du client, permettant de stocker des pages (propres ou modifiées) dans la mémoire d'un ou plusieurs « serveurs » (déterminés à l'avance) qui hébergent un démon utilisateur prévu à cet effet. Lorsqu'un serveur a besoin de mémoire, les pages à sauvegarder doivent, au préalable, être synchronisées sur la partition du client. La tolérance aux pannes est gérée grâce à des schémas de redondance classiques (RAID) ou à un mécanisme plus élaboré avec un tampon de parité maintenu au niveau du client et synchronisé périodiquement sur l'un des serveurs.
- Plus récemment, le système Nswap [159] a proposé une approche pair-à-pair pour la gestion d'un espace de pagination réparti. Chaque nœud héberge à la fois un module client et un module serveur au sein du noyau. Le module serveur régule dynamiquement la taille du cache de pages offert aux autres nœuds en fonction des besoins en mémoire des processus locaux. Un mécanisme de diffusion périodique permet à chaque serveur d'informer les clients actuels ou

[13] La plupart des travaux du domaine parlent, par abus de langage, de *swap* plutôt que de *pagination* mais il est, en fait, bel bien question de pagination (échanges à la granularité de la page de mémoire virtuelle et non à celle d'un espace virtuel complet).

potentiels des ressources disponibles afin d'aider ceux-ci dans dans leurs demandes d'allocation. Un serveur peut prendre l'initiative de déplacer des pages sur d'autres nœuds afin d'alléger sa charge. En dernier recours, les pages peuvent être enregistrées sur disque au niveau du client. La gestion de la tolérance aux pannes est rendue compliquée par le fait que les ressources des serveurs peuvent varier et car il faut empêcher les situations où, par le jeu de migrations, des informations redondantes sur une même page (copies intégrales ou informations de parité) se retrouvent sur le même nœud.

Nous proposons une solution intermédiaire, basé sur un compromis différent de ceux des solutions existantes. Nos choix sont résumés au travers des critères suivants :

Impact sur le code La refonte profonde des mécanismes de mémoire virtuelle n'est pas souhaitable, en raison de sa complexité. Ce type d'approche suscite, de plus, la méfiance des administrateurs, souvent peu enclins à accepter des modifications importantes aux systèmes d'exploitation qu'ils déploient. Une solution basée sur le chargement de modules noyaux est sensiblement moins contraignante. Nous rejoignons ainsi sur ce point l'approche de RMP.

Administration La possibilité de pouvoir potentiellement utiliser n'importe quel(s) nœud(s) d'une grappe comme serveur(s) de pages est intéressante. Cependant, un algorithme de gestion globale de la mémoire n'est pas bien adapté à la majorité des grappes actuelles qui sont souvent partitionnées dynamiquement pour gérér des tâches hétérogènes. En outre, une gestion totalement décentralisée à la Nswap présente une lacune importante à nos yeux : les décisions de migration sont prises uniquement en fonction d'une situation ponctuelle, sans corrélation avec les évolutions récentes et prévisibles de l'ensemble de nœuds.

Une piste intéressante semble être basée sur une prise de decision centralisée[14]. Les heuristiques employées pourraient s'appuyer sur les données recueillies par le service de supervision de la grappe ainsi que sur d'autres indicateurs. Par exemple, le système d'allocation/réservation de nœuds peut aider à prévoir l'évolution des ressources disponibles en fonction des échéances de réservation et du profil des tâches annoncées[15].

Flexibilité Nous proposons d'utiliser le concept de chemins d'E/S pour répondre aux besoins de flexibilité d'une infrastructure de pagination répartie. Grâce à la nature modulaire de Proboscis, il est possible de concevoir différentes solutions en fonction des ressources offertes par les serveurs. Un serveur peut héberger un simple ramdisk (disque virtuel de taille fixe), mettre à disposition un disque dur (et une portion de son cache de blocs) ou encore fournir un ensemble de zones mémoire dont la taille est susceptible de varier avec le temps.

Les deux premières approches permettent d'employer des stratégies simples de redondance (similaires à celles de RMP). De plus, l'emploi de mécanismes de reconfiguration (tel que celui détaillé en 6.5.2 pour un schéma de duplication complète) simplifient et diminuent le coût associé à la reconstruction de données suite à une panne ou à la migration de données sur un nouveau serveur (le client n'est pas impliqué dans les transferts de données entre serveurs).

[14] Par « centralisée », nous n'entendons pas forcément une entité de decision unique. Il pourrait s'agir d'un ensemble restreint de gestionnaires fortement couplés pour permettre davantage de passage à l'échelle.

[15] La mise en œuvre d'un tel système de réservation repose sur deux hypothèses fortes. D'une part, la possibilité pour les utilisateurs de spécifier des tâches « non perturbées », c'est-à-dire de restreindre ou d'interdire le partage de ressources avec d'autres nœuds (par exemple, dans le cas de mesures de performances). D'autre part, le maintient d'une base de données associant un programme (caractérisé par son nom et ses paramètres de lancement) à un ensemble de métriques de charge.

Cette proposition est ambitieuse et nous n'avons, par manque de temps, pas pu l'étayer davantage ni, *a fortiori*, la valider. Nous pensons néanmoins que pour commencer, une version simple d'un gestionnaire centralisé, reposant uniquement sur les informations de charge fournies par un service de supervision, pourrait s'avérer au moins aussi efficace que l'approche actuellement adoptée par Nswap.

La troisième approche laisse plus de souplesse pour gérer dynamiquement les ressources allouées par les serveurs mais nécessite le développement d'un module client spécialisé (plus complexe qu'un simple composant RAID), qui doit prendre en charge les migrations de pages entre serveurs[16] et pose les mêmes problèmes que NSwap pour la tolérance aux pannes.

En résumé, Proboscis permet à la fois d'implémenter une approche simple à la RMP, très proche du principe de RAID distribué mono-client, et une approche par migration de pages à la NSwap.

Nous décrivons maintenant notre implémentation actuelle. Le démon d'administration `pad` (cf. 6.1) s'exécutant sur un nœud contraint par une forte activité de pagination peut contacter le serveur central (SEC, cf. 6.1)pour demander des ressources supplémentaires. En réponse, le SEC retourne une description de chemin vers un ou plusieurs autres nœuds faisant office de serveurs de pagination[17]. Le démon `pad` déploie alors le chemins vers le(s) serveur(s) et enregistre le PGB correspondant comme une nouvelle partition de swap auprès du système d'exploitation.

Au niveau d'un serveur, le stockage des données peut être réalisé par un ramdisk ou sur disque, avec utilisation du cache de blocs[18]. Dans les deux cas, les mécanismes d'IODSM permettent d'éviter les recopies, en écrivant les données directement à l'emplacement approprié en mémoire. Le stockage sur disque via le cache (*write-back*) est plus avantageux du point de vue d'un serveur : celui-ci peut dynamiquement modifier l'espace mémoire alloué au client (moyennant la latence de synchronisation sur disque des blocs récemment modifiés). Un cache *write-through* est moins intéressant pour les performances (en écriture) d'un client mais permet à un serveur de se réattribuer instantanément de la mémoire pour ses besoins propres.

Pour caractériser les performances de la pagination à distance, nous avons défini un jeu de tests (partiellement inspiré par celui décrit dans [5]) basé sur des manipulations de grandes matrices (d'une taille supérieure à celle de la mémoire physique). Ces tests consistent à initialiser la matrice ligne par ligne puis à la balayer (en lecture et/ou écriture) par colonne, de façon à maximiser le nombre de défauts de page. Ceci permet de mettre en évidence les performances du périphérique de swap car le système passe alors la quasi-totalité de son temps à paginer, avec une séquence d'accès aux pages reproductible.

Les expériences ont été effectuées avec les machines de type B sur Gigabit Ethernet, avec un seul serveur de pagination (non chargé) et des transferts de données couplés aux requêtes. Avec un cache *write-back* au niveau du serveur, le facteur d'accélération par rapport à un disque local (en temps d'exécution du test) est compris entre 4,3 et 9,4, avec une moyenne de 8,2. L'emploi d'un ramdisk à la place du cache ne fournit pas de gain significatif (accélération moyenne de 8,4) par rapport à la perte de flexibilité qu'il occasionne. Avec un cache *write-through*, les performances en écriture sont assez proches de celles d'un disque local (différence de 8%). Le facteur d'accélération est dans ce cas uniquement due aux lectures (dont les performances sont, comme attendu, identiques à celles du cache *write-back*) et se limite en moyenne à un facteur 2,7 pour nos tests. Enfin, pour une configuration fonctionnelle équivalente[19], les performances mesurées avec Proboscis sont, à 1% près, celles observées avec GNBD.

[16]Les pages déplacées d'un serveur à un autre n'ont cependant pas à transiter par le client, grâce à l'utilisation des mécanismes d'IODSM.

[17]En l'état actuel, le SEC dispose d'une simple liste de serveurs.

[18]Lorsque les serveurs sont utilisés en mode *cache*, une extension Proboscis est ajoutée (en amont d'un éventuel composant RAID) au niveau du client. Celle-ci déploie un thread assistant qui explore périodiquement les structures de données du système de mémoire virtuelle afin de détecter d'éventuelles pages « mortes ». Le cas échéant, une commande d'invalidation est envoyée aux serveurs concernés pour libérer les blocs devenus inutiles.

[19]GNBD offre deux modes de fonctionnement pour le serveur : cache write-back ou aucun cache.

7.7.2 Caches coopératifs

À l'instar des systèmes de pagination répartis, les mécanismes de caches coopératifs [60] reposent sur des idées relativement simples et ont été déclinés en différentes variantes. Pour les mêmes raisons que celles évoquées en 7.7.1, nous nous intéressons à une mise en œuvre qui n'implique pas de gestion globale de la mémoire au sein de la grappe. Ceci correspond à l'approche connue sous le nom de *greedy forwarding*, où l'ensemble des clients mettent en commun le contenu de leur caches respectifs, sans pour autant coordonner leurs décisions de remplacement d'entrées.

De manière plus précise, nous considérons une architecture où plusieurs nœuds mettent en commun leurs disques pour former un SAN virtuel, exploité par le biais d'un SGF partagé. Chaque nœud joue donc à la fois le rôle de client et de serveur. En tant que client, un nœud dispose d'un cache de blocs (*buffer cache*). Le SGF partagé assure la mise en cohérence des différents caches clients. Le cache de blocs peut également être utilisé par le serveur afin de réduire la latence des accès effectués par les clients distants. Cependant, ceci peut conduire à une situation contre-productive dans la mesure où la capacité du cache d'un nœud (pour ses propres besoins) est restreinte par l'ajout des blocs utilisés par les clients qu'il sert, et d'autre part, la capacité mémoire globale de la grappe est quelque peu gaspillée car un même bloc est généralement présent à la fois dans le cache du client et celui du serveur. Par opposition, un mécanisme de caches coopératifs permet un usage plus harmonieux de la mémoire des serveurs.

Notre proposition vise à intégrer simplement une fonctionnalité de *greedy forwarding* au sein d'un SAN virtuel. Cet objectif est particulièrement complémentaire avec les responsabilités d'un SGF partagé (tel que GFS), qui gère les aspects de cohérence des caches, de verrouillage et de tolérance aux pannes mais ne fournit, dans la plupart des cas, aucun mécanisme de cache coopératif entre les nœuds.

Nous allons maintenant décrire plus en détails l'implémentation réalisée avec Proboscis. Son fonctionnement est résumé sur la figure 7.14.

Lorsqu'un client (n°1) envoie une requête de lecture au serveur, celle-ci est transmise jusqu'à l'extension `track` (cf. flèche n°1 sur le schéma) qui commence par vérifier si les blocs demandés sont disponibles dans le cache local. Si les blocs ont été trouvés, alors ceux-ci sont transférés au client et la requête est acquittée. En revanche, si les blocs ne sont pas disponibles, l'extension `track` consulte une table interne où elle maintient des informations sur les blocs récemment lus par les différents clients. Si un client (n°2) est trouvé par ce biais, alors une requête de coopération lui est transmise (flèche n°2). Cette requête, qui englobe une copie du *buffer aggregate* (BA) associé à la commande initiale de lecture, est traitée par l'extension `coop` au niveau du client interrogé. Celle-ci commence par vérifier la présence des blocs au sein du cache local. Le cas échéant, elle les verrouille et transmet leur contenu au client (n°1) qui les a demandés, via le mécanisme d'IODSM (flèche n°3). Si les blocs ne sont plus dans le cache, un fanion est activé au sein de la commande. Dans tous les cas, la commande est acquittée et revient au niveau de l'extension `track` sur le serveur (flèche n°4). Si les blocs n'ont pas été trouvés sur le client interrogé, la requête de lecture initiale est transmise au disque local (et les données lues ne seront pas mises en caches), sinon elle est acquittée immédiatement (flèche n°5).

Remarque : Si le chemin comporte, au niveau du client interrogé, une extension de dispersion qui agit sur l'adressage des données, celle-ci doit modifier les méta-données des requêtes de coopération qu'elle reçoit, grâce à la méthode de conversion d'adresses prévue à cet effet (cf. 4.2.4.4). Ceci est nécessaire car l'interrogation du *buffer cache* par l'extension `coop` doit être basée sur les adresses visibles à la racine du chemin et non sur celles utilisées à l'autre extrémité, au niveau du disque.

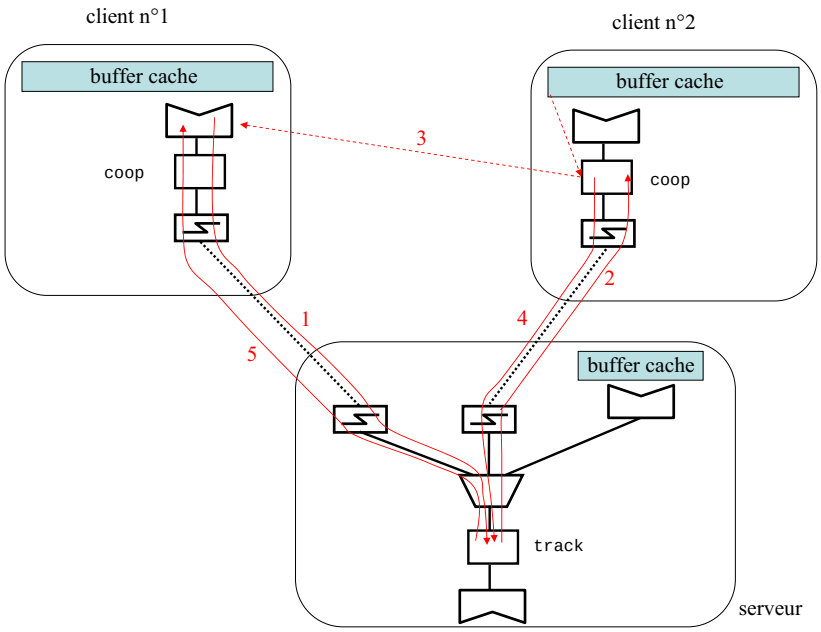


Figure 7.14 – Mise en œuvre d'un système de cache coopératif avec Proboscis

Le schéma de communication employé n'est pas optimal, en raison des contraintes imposées par Proboscis : une commande doit passer par le même chemin à l'aller et au retour. Ainsi, quatre communications réseau (plus le transfert de données proprement dit) sont nécessaires alors que trois messages pourraient suffire. Cependant, les données sont transférées directement entre les deux clients et les commandes échangées sont de petite taille (quelques centaines d'octets au plus) et induisent donc une faible latence.

L'implémentation actuelle est relativement naïve. Les informations maintenues par l'extension `track` sont conservées dans une table statique où chaque bloc est associé à l'identificateur du dernier client qui l'a lu ou modifié (ce qui ne permet qu'une seule tentative de lecture en cache distant). Des améliorations sont bien sûr envisageables (gestion dynamique des entrées, fusion des informations lorsque plusieurs blocs adjacents sont disponibles auprès du même client, etc.) mais même dans cette configuration basique, le coût en mémoire reste acceptable. Le ratio entre la taille mémoire nécessaire à l'extension `track` et l'espace de stockage disponible sur disque correspond à 0,1% pour une taille de bloc de 4 ko et à 0,0125% pour une taille de bloc de 32 ko (soit respectivement 100 Mo et 12 Mo de mémoire pour un disque de 100 Go).

Les performances de notre prototype sur Gigabit Ethernet (machine de type B) sont représentées en figure 7.15. Les quatre courbes correspondent aux cas suivants :

1. *cache serveur* : les données demandées par le client sont présentes dans le cache du serveur ;
2. *cache coop* : les données ne sont pas présentes dans le cache du serveur mais sont fournies par le cache d'un autre client ;
3. *disque serveur* : les données ne sont pas présentes dans le cache du serveur et aucune trace d'accès par un autre client n'est disponible auprès de l'extension `track`, elles doivent donc être lues sur le disque du serveur ;

4. *échec cache coop* : l'interrogation d'un client a échoué (les informations maintenues par l'extension `track` n'étant qu'approximatives) et il faut finalement lire les données sur le disque du serveur.

L'écart de latence entre les cas 1 et 2 n'est pas très important (130 à 290 μs) et son ratio décroît avec la taille des requêtes (pour atteindre 20% à 72 ko et 14% à 124 ko. En outre, l'écart de latence entre les cas 3 et 4 est faible également faible (250 à 600 μs, soit 4% à 8%). Enfin, comparé à un accès au disque sur le serveur, la lecture de données dans un cache distant est cinq à dix fois plus rapide.

Dans les cas où le client interrogé ne dispose pas d'une connexion ouverte avec le client demandeur des données, l'établissement d'une connexion nécessite en moyenne 80μs suppléméntaires[20]. Ceci ne diminue pas significativement l'amélioration apportée par le cache réparti par rapport à un accès disque. Seul l'écart de performances par rapport au cache du serveur est sensiblement accentué (latence augmentée en moyenne de 36% contre 25% si la connexion est déjà établie).

Ainsi, du point de vue du client émetteur de la requête de lecture, le système de cache coopératif offre un potentiel intéressant pour améliorer les performances d'accès aux données, et ce, même avec un réseau peu efficace tel que Gigabit Ethernet.

Les mêmes expériences sur SCI montrent qu'une interface de communication optimisée permet de réduire la latence des accès à distance et surtout de réduire l'impact du système coopératif sur le serveur et le client interrogé en diminuant le coût associé aux communications (et en supprimant éventuellement la nécessité d'établir des connexions entre clients).

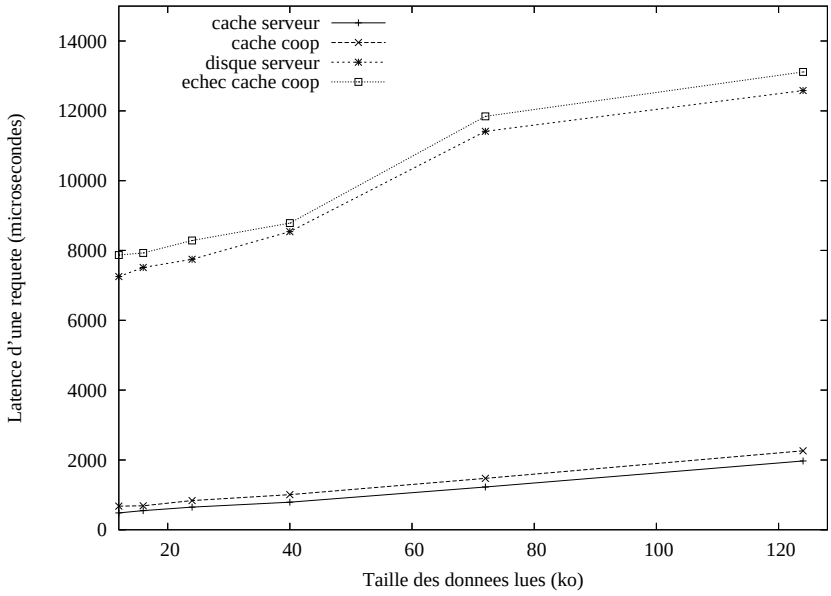


Figure 7.15 – Performance du système de cache coopératif avec Proboscis sur Gigabit Ethernet

7.8 Synthèse

Ce chapitre a présenté une évaluation de l'infrastructure de stockage réparti que nous avons proposée.

[20] Voir 5.4.2 pour plus de détails sur le sujet.

Nous avons, d'une part, cherché à évaluer la complexité induite par le modèle de programmation associé à notre infrastructure. Pour une fonction donnée, notre approche engendre généralement une augmentation de 10 à 30% du volume de code. Elle permet en revanche de clarifier le code d'un service de stockage en isolant clairement les aspects liés aux communications et en forçant les programmeurs à spécifier son fonctionnement sous la forme d'un automate. La programmation par événements est certes complexe à appréhender mais, au fil de notre implémentation, nous avons pu remarquer que les principaux bogues rencontrés n'étaient pas liés à ce modèle précis mais à des problèmes classiques de la programmation système (synchronisation, mauvais pointeurs, etc.).

En réalisant deux services de stockage spécialisés (pagination répartie et caches coopératifs), nous avons pu vérifier que la réutilisation de code est bien rendue possible par l'architecture et la granularité fonctionnelle choisies pour les composants de notre canevas.

Les expériences effectuées ont pour l'instant été limitées au stade des « micro-benchmarks ». Les résultats nous amènent néanmoins à conclure que l'approche décrite dans ce manuscrit est viable du point de vue des performances. La mise en œuvre d'un service de stockage à partir de composants logiciels programmés de manière asynchrone et associés à un contexte d'exécution particulier n'entraîne pas de dégradation sensible au niveau de la vitesse d'accès aux données ni à celui des ressources consommées. La représentation explicite des connexions entre composants répartis et l'emploi d'un modèle de programmation flexible pour la gestion des transferts de données ne diminuent pas non plus fortement les performances. Enfin, les protocoles de reconfiguration dynamique, rendus possibles par la souplesse du modèle introduit, sont compatibles avec une lourde charge d'E/S injectée en concurrence.

Par ailleurs, des tests « préliminaires » montrent que les optimisations matérielles au niveau des processeurs (SMT, SMP/CMP) et des périphériques de communication (jumbo frames Ethernet, RDMA) peuvent limiter de manière significative la charge induite sur un nœud jouant le rôle de serveur de stockage. Ainsi, il devient envisageable de répartir la fonction de serveur sur l'ensemble des nœuds d'une grappe.

La prochaine étape de validation reposera sur le déploiement d'applications réparties réalistes, avec une forte charge d'E/S, sur une grappe exploitée par des chemins Proboscis.

Conclusion

Après un rappel de nos principaux résultats, nous présentons quelques prolongements immédiats puis des perspectives à plus long terme.

8.1 Bilan

Le service de stockage réparti sur lequel repose une grappe de serveurs joue un rôle clé dans le fonctionnement global du système. Il doit en effet assurer la persistance des données et correspond à un aspect critique en performances car il implique deux types d'interactions parmi les plus coûteuses en ressources et en temps : les communications à distance et le dialogue avec des organes de stockage (dont la réactivité est limitée par des contraintes micro-mécaniques).

Notre étude bibliographique du domaine témoigne des nombreux et perpétuels efforts d'optimisation au niveau des performances d'accès aux données et des capacités de tolérance aux pannes. Elle illustre également la grande diversité de compromis adoptés entre les différents paramètres (pré-requis matériels, topologie, sémantiques de partage et de cohérence, simplicité de mise en œuvre, etc.) en fonction des besoins variés des applications concernées.

Il est également patent que la plupart des solutions proposées à ce jour ont été construites de manière relativement *ad hoc* et n'offrent pas ou peu de souplesse. Ainsi, dans la plupart des cas, le mécanisme de déploiement est figé, il n'est pas possible d'adapter finement le système à la plate-forme cible, et, *a fortiori*, d'agir sur celui-ci en cours de fonctionnement pour réagir à l'évolution des contraintes qu'il doit gérer. Ces limites s'avèrent d'autant plus problématiques que la haute disponibilité du service de stockage est généralement un postulat nécessaire pour la reconfiguration dynamique des autres couches logicielles. En outre, du fait des structures monolithiques généralement adoptées, il est difficile de réutiliser du code existant pour simplifier le développement d'un nouveau système. Enfin, nous considérons aussi le manque de flexibilité et de modularité des implémentations comme un obstacle important au développement de capacités d'auto-administration.

Au cours de cette thèse, nous avons donc cherché à étendre la flexibilité des systèmes de stockage pour grappes en proposant des modèles d'architecture logicielle. Nous avons veillé à définir des approches génériques. En particulier, nous n'avons pas fait d'hypothèses fortes sur les fonctionnalités disponibles au niveau des périphériques de communication et de stockage, ni à celui du système d'exploitation.

Nous avons commencé par définir un canevas extensible basé sur des composants logiciels, permettant de représenter explicitement et de configurer de manière fine (y compris au niveau des protocoles réseau employés) la chaîne de traitement que traversent les requêtes échangées entre une application

et un ensemble d'organes de stockage. En complément, une infrastructure d'exécution fournit les services nécessaires pour l'enregistrement de nouveaux types de composants, le déploiement de chemins et l'allocation des ressources physiques nécessaires à leur fonctionnement.

Nous avons également proposé un modèle de programmation permettant de spécifier et d'optimiser, de manière orthogonale au code de l'application, la façon dont deux nœuds coopératifs s'échangent des tampons de données. Il est ainsi possible d'adapter, de manière statique ou dynamique, la stratégie de transfert employée en fonction de différents paramètres tels que les technologies d'interconnexion disponibles, la topologie de l'application ou encore les contraintes de charge sur les différents nœuds.

Le principe de base consiste à gérer différemment deux types distincts de communications et, plus précisément, à découpler les messages de contrôle des échanges de données. Cette technique peut aider à produire du code plus générique pour les serveurs de données. Dans certaines conditions, elle contribue également à une meilleure gestion des tampons d'E/S en évitant des recopies inutiles et en diminuant leur durée d'immobilisation.

Enfin, pour fournir un point central de supervision et d'action aux agents humains responsables du service de stockage, nous avons défini une infrastructure d'administration spécialisée pour l'abstraction de *chemins d'E/S* et basée sur un serveur d'état. Celle-ci intègre des mécanismes de reconfiguration dynamique qui permettent d'ajouter de nouveaux types de composants, de modifier les paramètres ou le code d'une instance ou encore d'agir sur la structure d'un chemin sans interrompre le fonctionnement global du système de stockage. Ces fonctionnalités de base ne nécessitent pas ou peu de prise en charge au niveau du code des composants mais peuvent servir de fondations pour le développement de protocoles de reconfiguration plus complexes.

Un prototype a été développé au niveau d'interface blocs pour évaluer la viabilité d'une approche à composants dans le contexte des disques virtuels répartis, de la pagination à distance et des caches coopératifs, en considérant à la fois les réseaux banalisés tels que Gigabit Ethernet et les réseaux plus perfectionnés tels que SCI (accès à la mémoire distante par DMA ou couplage local).

Les expériences ont mis en évidence un coût acceptable, à la fois pour l'introduction d'une architecture à composants (0 à 2%) et pour l'emploi d'une infrastructure flexible pour les échanges de données (1 à 4%). Nous avons également montré l'intérêt d'un découplage potentiel entre les messages de contrôle et les transferts de données dans certaines situations (diminution de la charge mémoire d'un serveur, suppression des recopies de données sur les nœuds routeurs). Enfin, nous avons vérifié la compatibilité des protocoles de reconfiguration avec une forte charge d'E/S injectée en concurrence.

Nous avons donc proposé plusieurs briques logicielles de base pour la construction d'infrastructures de stockage efficaces, configurables et hautement disponibles. Cette contribution vise un grand spectre de contextes d'applications. Elle peut simplifier la mise en œuvre d'un système à image unique mais ne repose sur aucun pré-requis de ce type et convient également à une ferme de serveurs de données dont les systèmes d'exploitation sont moins fortement couplés. Par ailleurs, si nous avons principalement étudié des configurations où l'ensemble des nœuds partagent leurs ressources entre l'application et le service de stockage réparti, les mêmes principes peuvent s'appliquer à une grappe exclusivement consacrée à ce second rôle.

8.2 Perspectives proches

8.2.1 Implémentation à différents niveaux d'interface

Le prototype que nous avons développé est pour l'instant restreint à l'interface blocs. Pour évaluer notre proposition de manière plus globale, il est important de poursuivre l'implémentation du canevas, en ajoutant des composants à d'autres niveaux d'interface, en particulier au niveau « fichiers ». Compte tenu des résultats encourageants déjà obtenus par d'autres projets de recherche sur les systèmes de gestion de fichiers modulaires, nous sommes optimistes quant aux performances envisageables. Ces expérimentations auraient toutefois l'intérêt complémentaire de nous aider à mieux jauger la difficulté introduite par le modèle de programmation propre à notre canevas.

La prise en charge du niveau d'interface fichiers permettrait, d'une part, d'implémenter des systèmes client-serveur fonctionnellement équivalents à ceux présentés en 2.4.2.4 mais avec davantage de flexibilité au niveau des mécanismes de transfert. D'autre part, la réalisation d'un SGF partagé à partir de composants contribuerait probablement à mettre en évidence et à combler d'éventuelles lacunes du modèle de composants actuellement employé.

8.2.2 Optimisations plus avancées

Une piste intéressante pour l'étude des performances consisterait à déterminer les gains possibles avec une approche moins générique. La version actuelle du prototype a été conçue pour une intégration facile avec le système d'exploitation hôte (aux deux extrémités d'un chemin), et en particulier pour supporter n'importe quel type de périphérique orienté blocs. En conséquence, les requêtes d'E/S émises par un client sont d'abord soumises à l'ordonnanceur (de disque) local, puis transmises au PGB Proboscis qui les achemine jusqu'au(x) nœuds serveur(s), où elles sont à nouveau traitées par le système d'exploitation hôte.

D'autres compromis sont possibles, par le biais d'extensions d'extrémité plus spécialisées. On pourrait par exemple « court-circuiter » l'un ou l'autre des ordonnanceurs[21] en interceptant/réinjectant les requêtes d'E/S à plus bas niveau.

Dans le même ordre d'idées, on pourrait étudier en détails l'intérêt des protocoles de transport réseau optimisés à bas niveau comme HyperSCSI. Les tests préliminaires que nous avons pu mener laissent envisager des améliorations de performances substantielles par rapport à la pile TCP/IP mais un comportement plus erratique lorsque le réseau est chargé et, *a fortiori*, utilisé conjointement pour du trafic applicatif et du stockage de données.

8.2.3 Support plus fin pour outils de virtualisation de l'espace de stockage

Les couches de virtualisation pour systèmes de stockage (gestionnaires de volumes logiques, mécanismes de snapshot) reposent sur un recours important à des méta-données (tables de traduction d'adresses, etc.) dont la gestion n'est pas triviale (contraintes de volume mémoire, de synchronisation et de persistance). Pour simplifier la tâche des auteurs de tels outils, il serait avantageux d'intégrer des mécanismes prévus à cet effet au sein de l'infrastructure Proboscis. Les propositions développées récemment dans le cadre du canevas Violin [72] semblent constituer un bon point de départ.

[21] *A priori*, il semble plus avantageux de conserver l'ordonnanceur sur le nœud client si le périphérique de stockage n'est pas partagé (meilleure utilisation du réseau via des regroupements de requêtes) et sur le serveur dans le cas contraire (décisions d'ordonnancement à partir d'une vue complète des différents flux).

8.2.4 Intégration avec les machines virtuelles

Comme expliqué en 1.2.2.3, un nombre croissant de serveurs de données sont déployés au sein de machines virtuelles. Dans ce cas de figure, les ressources physiques d'un nœud sont multiplexées par un mini-système d'exploitation (moniteur) entre les différentes machines virtuelles qu'il héberge. Le partage de données entre les différents serveurs virtuels (à l'échelle locale comme à celle de la grappe) est possible par l'intermédiaire d'un SAN, exporté par chaque moniteur vers les machines virtuelles qu'il héberge.

L'emploi d'une infrastructure flexible comme Proboscis pour fournir le système de stockage partagé semble prometteur. Les capacités de reconfiguration dynamique permettraient notamment de répondre aux besoins motivés par la migration d'une machine virtuelle (déploiement de nouveaux chemins d'E/S depuis/vers le nœud de destination). Ces techniques pourraient également être appliquées à la migration de périphériques logiques non partagés (partition de swap, base de données locale, etc.)[22].

8.2.5 Système de clonage pour grappes

Une autre application directe de nos travaux concerne les systèmes de clonage pour grappes. Ce type d'application a pour rôle de recopier de manière fiable et efficace un même ensemble de données (généralement une image de système d'exploitation) sur le disque local d'un grand nombre de nœuds. Deux approches principales ont été développées à cet effet. La première est basée sur l'emploi d'UDP multicast [96]. Elle est bien adaptée au cas d'un réseau « à vide » (pas ou peu de trafic concurrent). La seconde repose sur un arbre/pipeline de connexions point-à-point fiables. Elle engendre des latences plus élevées à grande échelle, mais s'adapte mieux aux problèmes de charge réseau (potentiellement fréquents si seule une partie de la grappe est clonée) et offre une solution de repli en l'absence de support matériel pour la diffusion [19]. Chacune de ces stratégies peut éventuellement être combinée à des techniques de compression optimisées pour le type de contenu considéré.

Dans l'état actuel du prototype, Proboscis pourrait assez facilement permettre la construction d'une application de clonage par arbre/pipeline (avec support de la compression). Les caractéristiques flexibles du canevas seraient notamment avantageuses pour la prise en charge de grappes hétérogènes (différentes technologies d'interconnexion) et de contraintes dynamiques (ajout/retrait de nœuds en cours de clonage).

En modifiant le modèle de communication pour prendre en compte les communications par diffusion, il serait envisageable d'obtenir une application hybride, capable de basculer entre les deux stratégies en fonction de leurs performances respectives.

8.3 Champs d'investigation complémentaires

8.3.1 Cible de déploiement

L'infrastructure que nous proposons a été pensée comme une entité chargée au sein du noyau d'un système d'exploitation et notre implémentation prototype se conforme à cette idée, motivée à la fois par des critères d'efficacité et de généricité (tous les programmes/services peuvent ainsi utiliser des chemins pour accéder à des données). D'autres niveaux d'implémentation sont cependant envisageables ; nous en décrivons deux ci-dessous.

Implémentation de l'infrastructure au niveau utilisateur Il est tout à fait concevable d'implémenter l'infrastructure au niveau utilisateur. Celle-ci gagnerait alors en portabilité mais perdrait

[22] Voir [36], section 7.3 pour plus de détails sur ce point.

en généralité (utilisable uniquement par des applications liées à une bibliothèque adéquate). Cette approche est particulièrement adaptée pour les interfaces de communication capables d'*OS bypass* et s'apparente à une version plus modulaire de plates-formes telles que DAFS (cf 2.4.2.4).

Implémentation de l'infrastructure au sein d'une interface de communication Compte tenu :

- de la faible taille du code associé à l'infrastructure Proboscis,
- de la consommation mémoire modérée de celle-ci,
- de son caractère modulaire (qui permet de limiter le volume de code en ne conservant que les modules utiles pour un contexte applicatif précis),
- des ressources dont disposent les cartes réseau actuelles équipées d'un processeur reprogrammable,

il est envisageable d'embarquer une infrastructure dynamique telle que Proboscis au sein d'une interface de communication optimisée (Myrinet par exemple). Ceci permettrait d'obtenir une version plus flexible de systèmes comme $READ^2$ (cf. 2.3.1.2) qui permettent de décharger le processeur d'un nœud serveur.

Par ailleurs, des travaux récents ont proposé d'exploiter le processeur embarqué et les capacités de RDMA d'une carte réseau afin de permettre la surveillance à distance (et l'éventuelle réparation) de l'intégrité d'un système d'exploitation [28]. Proboscis nous semble constituer une base de départ intéressante pour une mise en œuvre modulaire de tels services.

8.3.2 Gestion globale du contrôle de flux et de la qualité de service

Dans l'état actuel de notre proposition, le contrôle de flux au sein d'un chemin n'est pas effectué de manière globale. Des mécanismes distincts, externes à Proboscis et non couplés, sont simplement mis en œuvre à différents niveaux, typiquement aux deux extrémités (ordonnanceur disque) ainsi qu'à chaque frontière réseau (via le protocole de transport employé). Dans certaines configurations, des mécanismes internes à Proboscis peuvent être utilisés en complément, par exemple au niveau d'une extension d'agrégation.

Cet état de fait n'est pas suffisant pour permettre une gestion optimale du trafic et associer d'éventuelles garanties de qualité de service à certains chemins. Ce problème est loin d'être trivial et l'état de l'art dans le domaine ne fournit pas de réponse directement transposable à nos préoccupations[23]. Parmi les leviers envisageables, on peut notamment citer le développement d'un ordonnanceur de tâches spécialisé (interne à Proboscis), le traitement par lots de certains types d'événements ou encore l'ajout d'informations/recommandations explicites de charge au sein des commandes acquittées pour instaurer une boucle de contrôle répartie.

8.3.3 Support optimisé pour architecture multiprocesseurs

Le premier chapitre a mis en évidence la tendance émergente en matière d'architecture des microprocesseurs, à savoir l'augmentation du support matériel pour le parallélisme d'exécution. À moyen terme, les machines disposeront d'au moins quatre (voire huit) unités d'exécution physiques par puce,

[23] La plupart des canevas analogues, tels que Click [66] ou SEDA [232] ont considéré cet aspect dans un cadre centralisé. Les études dans un contexte réparti sont moins nombreuses et ne visent qu'une partie du problème. Par exemple, l'environnement River [15] s'intéresse davantage à une répartition optimale du travail entre des producteurs et consommateurs hétérogènes plutôt qu'à un véritable contrôle de flux.

éventuellement complétées par du parallélisme d'instructions (SMT). Dans ce contexte, un usage optimal des ressources va passer par l'adaptation des systèmes d'exploitation et des intergiciels à cette « nouvelle donne » matérielle.

Au niveau de Proboscis, cela incite à abandonner la restriction à un seul thread d'exécution, et pose en conséquence plusieurs problèmes.

- Comment gérer au mieux l'association entre les différents threads d'exécution et l'ensemble des instances de composants déployées sur un nœud ?
- Comment continuer à garantir la sérialisation des événements au sein d'une instance donnée ?
- Comment assurer l'intégrité d'un chemin en cas de reconfiguration dynamique ?

Ces questions soulignent deux besoins principaux : un ordonnanceur de tâches prenant en compte de multiples threads d'exécution (et cherchant à minimiser les changements de contexte lors de la traversée d'un chemin par une commande) et des mécanismes de synchronisation pour assurer le bon fonctionnement de l'infrastructure. Des travaux tels que la version SMP du routeur Click [49] pourraient nous inspirer pour le premier point mais les solutions proposées ne sont pas toutes compatibles avec nos contraintes.

8.3.4 Aide au développement

Le modèle de programmation par événements est indéniablement complexe à appréhender. En outre, la perspective d'utiliser de multiples threads d'exécution en parallèle risque d'accroître la probabilité de bogues en tous genres et de problèmes de synchronisation en particulier. Nous pensons cependant que ce modèle de programmation correspond à la meilleure solution pour spécifier le comportement des composants et la prise en charge des opérations de reconfiguration dynamique. Aussi, pour bénéficier des avantages de cette approche sans pâtir de ses inconvénients, une approche prometteuse consiste à recourir à des techniques de vérification. De tels outils existent depuis longtemps pour les systèmes critiques mais n'ont que récemment été adaptés au contexte des systèmes d'exploitation « généralistes »[24]. Le développement d'outils de vérification spécialisés pour Proboscis (au niveau d'un composant comme à celui d'un chemin complet) apparaît comme une étape nécessaire au développement de systèmes de stockage fiables.

En amont, le développement de programmes basés sur un modèle événementiel peut aussi être facilité par la mise en évidence de leur flot de contrôle. La très récente bibliothèque Eel [57] propose des avancées intéressantes dans cette direction.

Par ailleurs, la programmation par composants peut, en dépit de ses nombreuses qualités, s'avérer fastidieuse en raison des contraintes de structuration qu'elle impose. Afin d'améliorer la productivité des programmeurs (et d'éviter les implémentations incomplètes), les techniques de génération automatique de code (ou du moins de squelettes) semblent pertinentes. Les résultats obtenus par d'autres équipes dans le contexte des SGF empilables nous confortent dans cette démarche.

8.3.5 Systèmes autonomes

Enfin, la construction de systèmes de stockage autonomes n'en est qu'à ses balbutiements. Nous avons veillé à intégrer les bases nécessaires à des capacités d'auto-administration (réflexivité et reconfiguration dynamique) au sein de notre proposition, mais nous sommes loin d'avoir abouti aux objectifs ambitieux d'un projet comme « Self-* » (cf. 2.6.2). Les principaux défis à venir vont consister à définir de manière plus précise la hiérarchie de contrôle faisant le lien entre les approches de haut niveau (relatives à l'intelligence artificielle : auto-apprentissage, inférence, etc.) et celles de bas

[24] Voir notamment les travaux de l'équipe de D. Engler à ce sujet : `http://www.stanford.edu/~engler/`.

niveau (relatives au fonctionnement du système, telles que la notre). Ceci nécessitera probablement la définition de nouvelles métriques (de performance, de disponibilité ou plus généralement de satisfaction) et de l'interface de contrôle associée ainsi que la mise en place d'un ensemble de boucles de réaction.

Liste des abréviations

AFS Andrew File System. voir 2.4.1.

ANSI T10 Comité technique de standardisation des interfaces SCSI de l'*American National Standards Institute*

ATM Asynchronous Transfer Mode.

BA Buffer Aggregate. voir 5.2.1

CMU Carnegie Mellon University (Pittsburgh, USA).

DAFS Direct Access File System. voir 2.4.2.4.

DVR Disque Virtuel Réparti. voir 2.3.

FC Fibre Channel. voir 2.2.

FCP Fibre Channel Protocol. voir 2.2.

FS File System. voir SGF.

GNBD Global Network Block Device. voir 2.3.1.1.

GVL Gestionnaire de Volumes Logiques. voir 2.1.2.2.

GVLG Gestionnaire de Volumes Logiques pour Grappe. voir 2.3.3.

IAS IODSM Address Space. voir chapitre 5

IODSM Distributed Shared Memory for I/O transfers. voir chapitre 5

JFS Journalized | Journaling File System. voir 2.1.2.3.

LAN Local Area Network.

L(S)FS Log Structured File System. voir 2.1.2.3.

MAN Metropolitan Area Network.

NBD Network Block Device. voir 2.3.

NFS Network File System. voir 2.4.2.1.

O(B)SD Object-Based Storage Devices. voir 2.1.2.2.

P(G)B Périphérique (Générique) (à) Blocs. voir 2.1.2.2.

P(G)BD Périphérique (Générique) (à) Blocs Distant. voir 2.3.

RAID Redundant Array of Independent Disks. voir 2.1.2.2.

RVSD Recoverable Virtual Shared Disk. voir 2.3.

SAN Storage Area Network. voir 2.2.

SEC Serveur d'État Central. voir 6.1.

SCI Scalable Coherent Interface. voir 1.3.2.2.

SCSI Small Computer System Interface. voir 2.1.2.1.

S(G)F Système (de Gestion) de fichiers. voir 2.1.2.3.

SNIA Storage Networking Industry Association.

VFS Virtual File System. voir 2.1.3.

VV Volume Virtuel. voir 2.3.

Bibliographie

[1] *Bonnie++ Benchmark*, available `www.coker.com.au/bonnie++`.

[2] *IOmeter Benchmark*, available `http://www.iometer.org`.

[3] *IOZone Benchmark*, available `http://www.iozone.org`.

[4] A. Acharya, M. Uysal, and J. Saltz, *Active disks : Programming Model, Algorithms and Evaluation*, Proceedings of the 8th Symposium on Architectural Support for Programming Languages and Operating Systems, ACM Press, 1998, pp. 81–91.

[5] A. Acquaviva, E. Lattanzi, and A. Bogliolo, *Power-aware network swapping for wireless palmtop pcs*, Proceedings of the IEEE Design Automation and Test in Europe Conference (DATE-04) (Paris, France), IEEE Computer Society, February 2004.

[6] K. Amiri, *Scalable and Manageable Storage Systems*, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA, December 2000.

[7] K. Amiri, G. A. Gibson, and R. A. Golding, *Highly Concurrent Shared Storage*, Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS) (Tapei, Taiwan), IEEE Computer Society, April 2000, pp. 298–307.

[8] K. Amiri, D. Petrou, G. Ganger, and G. Gibson, *Dynamic function placement for data-intensive cluster computing*, Proceedings of the USENIX Annual Technical Conference (San Diego, CA, USA), USENIX Association, June 2000.

[9] D. Anderson, J. Dykes, and E. Riedel, *More than an interface – SCSI vs. ATA*, Proceedings of the 2nd Annual Conference on File and Storage Technology (FAST) (San Francisco, CA, USA), USENIX Association, March 2003.

[10] D. C. Anderson, J. Chase, and A. Vahdat, *Interposed request routing for scalable network storage*, Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI), USENIX Association, 2000.

[11] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, *Serverless Network File Systems*, ACM Transactions on Computer Systems **14** (1996), no. 1, 41–79.

[12] J. Appavoo, M. Auslander, M. Burtico, D. Da Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis, *K42 : an Open-Source Linux-Compatible Scalable Operating System Kernel*, IBM Systems Journal (2005), to appear.

[13] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis, *K42 Overview*, 2002, Available at `http://www.research.ibm.com/K42/white-papers/Overview.pdf`.

[14] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, and M. Kalantar, *Oceano : SLA Based Management of a Computing Utility*, Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management (New York City, NY, USA), IEEE Computer Society, May 2001.

[15] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick, *Cluster I/O with River : Making the fast case common*, Proceedings of the 6th Workshop on Input/Output in Parallel and Distributed Systems (IOPADS) (Atlanta, GA, USA), ACM Press, May 1999, pp. 10–22.

[16] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson, *The Architectural Costs of Streaming I/O : a Comparison of Workstations, Clusters, and SMPs*, Proceedings of the 4th Symposium on High-Performance Computer Architecture (HPCA-4) (Las Vegas, NV, USA), IEEE Computer Society, February 1998, pp. 90–101.

[17] S. Asami, *Reducing the Cost of System Administration of a Disk Storage System Built from Commodity Components*, Ph.D. thesis, University of California, Berkeley, Berkeley, CA, USA, 2000, Technical Report CSD-00-1100.

[18] C. R. Attanasio, M. Butrico, C. A. Polyzois, S. E. Smith, and J. L. Peterson, *Design and Implementation of a Recoverable Virtual Shared Disk*, Tech. Report Research Report RC 19843, IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, 1994.

[19] P. Augerat, W. Billot, S. Derr, and C. Martin, *A Scalable File Distribution and Operating System Installation Toolkit for Clusters*, Tech. report, Laboratoire ID-Imag / Inria, Grenoble, France, October 2001, available at `ka-tools.sourceforge.net/publications/file-distribution.pdf(2004-06-07)`.

[20] O. Aumage, L. Bougé, J. F. Méhaut, and R. Namyst, *Madeleine II : A Portable and Efficient Communication Library for High-Performance Cluster Computing*, Parallel Computing **28** (2002), no. 4, 607–626.

[21] S. Baker and J. H. Hartman, *The Mirage NFS Router*, Tech. Report TR02-04, University of Arizona, Tucson, AZ, USA, 2002.

[22] L. A. Barroso, J. Dean, and U. Hölzle, *Websearch for A Planet : The Google Cluster Architecture*, IEEE Micro **23** (2003), no. 2, 22–28.

[23] A. Barry, K. Preslan, M. O'Keefe, G. Johnsen, J. Wayda, and B. Alting, *SCSI Device Memory Export Protocol Version 0.9.8*, August 2000, Available at `http://www.t10.org/ftp/t10/document.00/00-312r0.pdf`.

[24] A. Baumann, G. Heiser, J. Appavoo, D. Da Silva, O. Krieger, R. W. Wisniewski, and J. Kerr, *Providing Dynamic Update in an Operating System*, Proceedings of the 2005 USENIX Technical Conference (Anaheim, CA, USA), USENIX Association, April 2005.

[25] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, *Extensibility, safety and performance in the SPIN operating system*, Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15) (Copper Mountain Resort, CO, USA), ACM Press, December 1995.

[26] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan, *Asynchronous I/O Support in Linux 2.5*, Proceedings of the 2003 Linux Symposium, July 2003.

[27] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su, *Myrinet : A gigabit-per-second Local Area Network*, IEEE Micro **15** (1995), no. 1, 29–36.

[28] A. Bohara, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode, *Remote Repair of Operating System State Using Backdoors*, Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04) (New York, NY, USA), IEEE Computer Society, May 2004.

[29] J. Bonwick, *The Slab Allocator : An Object-Caching Kernel Memory Allocator*, Proceedings of the Usenix Summer 1994 Technical Conference (Boston, MA, USA), USENIX Association, June 1994.

[30] A. J. Borr, *SecureShare : Safe UNIX/Windows File Sharing trough Multiprotocol Locking*, Proceedings of the 2nd USENIX Windows NT Symposium (Seattle, WA, 1998), USENIX Association, August 1998.

[31] P. J. Braam, *The Lustre Storage Architecture*, August 2004, Available at `http://www.lustre.org/docs/lustre.pdf` (2004-09-06).

[32] P. J. Braam and P. A. Nelson, *Removing Bottlenecks in Distributed Filesystems : Coda & InterMezzo as examples*, Proceedings of the 1999 Linux Expo (Raleigh, NC, USA), May 1999.

[33] P. T. Breuer, A. Marín Lopez, and A. García Ares, *The Network Block Device*, Linux Journal (2000), no. 73, available at `http://www.linuxjournal.com/article.php?sid=3778` (2004-03-08).

[34] R. Brightwell, R. Riesen, B. Lawry, and A. B. Maccabe, *Portals 3.0 : Protocol Building Blocks for Low Overhead Communication*, Proceedings of the Workshop on Communication Architecture for Clusters (CAC 2002) (Fort Lauderdale, Florida, USA), IEEE Computer Society, April 2002.

[35] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, *Exploiting Gray-Box Knowledge of Buffer-Cache Management*, Proceedings of the General Track : 2002 USENIX Annual Technical Conference, USENIX Association, 2002, pp. 29–44.

[36] C. Clark and K. Fraser and S. Hand and J. Gorm Hansen and E. Jul and C. Limpach and I. Pratt and A. Warfield, *Live Migration of Virtual Machines*, Proceedings of the 2nd Symposium on Network Systems Design and Implementation (Boston, MA, USA), USENIX Association, May 2005.

[37] B. Callaghan, T. Lingutla-Raj, A. Chiu, P. Staubach, and O. Asad, *NFS over RDMA*, Proceedings of the SIGCOMM Workshop on Network-I/O Convergence : Experience, Lessons, Implications (NICELI) (Karlsruhe, Germany), ACM Press, August 2003.

[38] B. Callaghan, B. Pawlowski, and P. Staubach, *NFS Version 3 Protocol*, RFC 1813, June 1995.

[39] B. Callaghan, D. Robinson, R. Thurlow, C. Beaume, M. Eisler, and D. Noveck, *NFS Version 4 Protocol*, RFC 3010, November 2000.

[40] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur, *PVFS : A parallel file system for linux clusters*, Proceedings of the 4th Annual Linux Showcase and Conference (Atlanta, GA, USA), USENIX Association, 2000.

[41] E. Caron, *Calcul numérique sur données de grande taille*, Ph.D. thesis, Université de Picardie Jules Verne, December 2000, Available at `http://graal.ens-lyon.fr/~ecaron/papers/these_caron.ps.gz`.

[42] E. Cecchet, *Apport des Réseaux à Capacité d'Adressage pour des Grappes à Mémoire Partagée Distribuée Logicielle - Conception et Applications*, Ph.D. thesis, INPG, Grenoble, France, July 2001.

[43] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, *Performance Comparison of Middleware Architectures for Generating Dynamic Web Content*, Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference (Middleware 2003) (Rio de Janeiro, Brazil), Spinger-Verlag, June 2003.

[44] E. Cecchet, J. Marguerite, and W. Zwaenepoel, *Partial Replication : Achieving Scalability in Redundant Arrays of Inexpensive Databases*, Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS'2003) (La Martinique, France), December 2003.

[45] _____, *C-JDBC : Flexible Database Clustering Middleware*, Proceedings of USENIX Annual Technical Conference, Freenix track (Boston, MA, USA), USENIX Association, June 2004.

[46] M. Chadapalaka, J. Hufferd, J. Satran, and H. Stah, *Datamover Architecture for iSCSI (version 1.0)*, 2003, RDMA Consortium, available at `http://www.rdmaconsortium.org/home/draft-chadalapaka-iwarp-da-v1.0.pdf`.

[47] J. Chase, D. Irwin, L. Grit, J. Moore, and S. Sprenkle, *Dynamic Virtual Clusters in a Grid Site Manager*, Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03) (Seattle, Washington, USA), IEEE, June 2003, pp. 90–103.

[48] J. S. Chase, D. C. Anderson, P. N. Thakar, A. Vahdat, and R. P. Doyle, *Managing Energy and Server Resources in Hosting Centres*, Proceedings of the 18th ACM Symposium on Operating Systems Principles (Banff, Alberta, Canada), October 2001, pp. 103–116.

[49] B. Chen and R. Morris, *Flexible Control of Parallelism in a Multiprocessor PC Router*, Proceedings of the 2001 USENIX Annual Technical Conference (USENIX '01) (Boston, Massachusetts, USA), USENIX Association, June 2001, pp. 333–346.

[50] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, *RAID : High-Performance, Reliable Secondary Storage*, ACM Computing Surveys **26** (1994), no. 2, 145–185.

[51] N. Christenson, T. Bosserman, and D. Beckemeyer, *A Highly Scalable Electronic Mail Service Using Open Systems*, Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS) (Monterey, CA, USA), December 1997.

[52] S. Chutami, O. Anderson, M. Kazer, B. Leverett, W. A. Mason, and R. Sidebotham, *The episode file system*, Proceedings of the 1992 Winter USENIX Technical Conference (San Francisco, CA, USA), USENIX Association, January 1992, pp. 43–60.

[53] G. Ciaccio and G. Chiola, *GAMMA and MPI/GAMMA on Gigabit Ethernet*, in Proceedings of the 7th EuroPVM-MPI Users Group Meeting (Balatonfüred, Hungary), vol. 1908, Springer Verlag, September 2000.

[54] Cluster File Systems Inc., *Lustre : a High-Performance, Scalable, Open Distributed File System for Clusters and Shared-Data Environments*, white paper, November 2002, Available at `http://www.lustre.org/docs/whitepaper.pdf` (2004-09-06).

[55] O. Cozette, C. Randriamaro, and G. Utard, *Improving Cluster IO Performance with Remote Efficient Access to Distant Device*, Proceedings of the 27th Annual Conference on Local Computer Networks (LCN'02) (Tampa, Florida, USA), IEEE Computer Society, November 2002.

[56] _____, *READ2 : Put disks at network level*, Workshop on Parallel I/O, Proceedings of the 3rd International Symposium on Cluster Computing and the Grid (Tokyo, Japan), IEEE Computer Society, May 2003, pp. 698–704.

[57] R. Cunningham and E. Kohler, *Making Events Less Slippery with Eel*, Proceedings of the 10th Workshop on Hot Topics in Operating Systems Symposium (HotOS-X) (Santa Fe, NM, USA), USENIX Association, June 2005.

[58] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazieres, and R. Morris, *Event-Driven Programming for Robust Software*, Proceedings of SIGOPS European Workshop 2002 (Saint-Emilion, France), September 2002.

[59] DAFS Collaborative, *the Direct Access File System Protocol*, See `http://www.dafscollaborative.org` for details.

[60] M. D. Dahlin, T. E. Anderson, D. A. Patterson, and R. Y. Wang, *Cooperative Caching : Using Remote Client Memory to Improve File System Performance*, Proceedings of the 1st Symposium on Operating Systems Design and Implementation (Monterey, CA, USA), USENIX Association, November 1994, pp. 267–280.

[61] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh, *The Logical Disk : A New Approach to Improving File Systems*, Proceedings of the 14th ACM Symposium on Operating Systems Principles (Asheville, NC, USA), ACM Press, December 1993, pp. 15–28.

[62] A Di Marco, G. Chiola, and G. Ciaccio, *Using a Gigabit Ethernet Cluster as a Distributed Disk Array with Multiple Fault Tolerance* , Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks (LCN) (Bonn/Königswinter, Germany), IEEE Computer Society, October 2003.

[63] R. Draves, B. Bershad an R. Rashid, and R. Dean, *Using Continuations to Implement Thread Management and Communication in Operating Systems*, Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP) (Pacific Grove, CA, USA), ACM Press, October 1991.

[64] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, *The Virtual Interface Architecture*, IEEE Micro **18** (1998), no. 2.

[65] M. Dydensborg, *Direct Remote Access to Devices*, Proceedings of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS), May 2001.

[66] E. Kohler and R. Morris and B. Chen and J. Jannotti and M. F. Kaashoek, *The Click Modular Router*, ACM Transactions on Computer Systems **18** (2000), no. 3, 263–297.

[67] K. Elmeleegy, A. Chanda, A. Cox, and W. Zwaenepoel, *Lazy Asynchronous I/O for Event-Driven Servers*, Proceedings of the USENIX 2004 Technical conference (Boston, MA, USA), USENIX Association, June 2004, pp. 241–254.

[68] J. P. Fassino, J. B. Stefani, J. Lawall, and G. Muller, *THINK : A Software Framework for Component-based Operating System Kernels*, Proceedings of Usenix Annual Technical Conference (Monterey, CA, USA), USENIX Association, June 2002.

[69] M. J. Feeley, W. E. Morgan, E. P. Pighin, A. R. Karlin, H. M. Levy, and C. A. Thekkath, *Implementing global memory management in a workstation cluster*, Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), ACM Press, 1995, pp. 201–212.

[70] W. Feng, M. Warren, and E. Weigle, *The Bladed Beowulf : A Cost-Effective Alternative to Traditional Beowulfs*, in Proceedings of IEEE Cluster 2002 (Chicago, Illinois, USA), IEEE Computer Society, September 2002, pp. 245–254.

[71] G. Finn, *An Integration of Network Communication with Workstation Architecture*, ACM Computer Communication Review **21** (1991), no. 5, 18–29.

[72] M. D. Flouris and A. Bilas, *Violin : A Framework for Extensible Block-Level Storage*, Proceedings of the 13th NASA Goddard and 22nd IEEE Conference on Mass Storage Systems and Technologies (MSST2005) (Monterey, CA, USA), IEEE Computer Society, April 2005.

[73] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann, *Interface and Execution Models in the Fluke Kernel*, Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI) (New Orleans, LA, USA), USENIX Association, February 1999.

[74] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, *Safe Hardware Access with the Xen Virtual Machine Monitor*, Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (Boston, MA, USA), ACM Press, October 2004.

[75] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch, *FAB : Enterprise Storage Systems on a Shoestring*, Proceedings of the 9th Workshop on Hot Topics in Operating Systems (Lihue, Hawai, USA), USENIX Association, May 2003.

[76] ———, *A Decentralized Algorithm for Erasure-Coded Virtual Disks*, Proceedings of the 2004 Conference on Dependable Systems and Networks (DSN 2004) (Florence, Italy), 2004.

[77] B. Gaffey, *CXFS : A General Purpose Cluster File System*, Proceedings of the 8th NASA Goddard Conference on Mass Storage Systems and Technologies (College Park, MD, USA), March 2000.

[78] G. R. Ganger, J. D. Strunk, and A. J. Klosterman, *Self-* Storage : Brick-based Storage with Automated Administration*, Tech. Report CMU-CS-03-178, Parallel Data Lab - Carnegie-Mellon University, August 2003, available at `http://www.pdl.cmu.edu/PDL-FTP/SelfStar/selfstar.pdf` (2004-03-08).

[79] P. Geoffray, *Contributions au support de communication des serveurs vidéo distribués suivant une architecture réseau de type grappe de PC*, Ph.D. thesis, Université Claude Bernard, Lyon, France, June 2001.

[80] ———, *OPIOM : Off Processor IO with Myrinet*, Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGrid 2001) (Brisbane, Australia), IEEE Computer Society, May 2001.

[81] S. Ghemawat, H. Gobioff, and S. Leung, *The Google file system*, Proceedings of the 19th ACM Symposium on Operating Systems Principles, ACM Press, 2003, pp. 29–43.

[82] G. Gibson, *Storage : 10 year retrospective, 10 year prospective*, invited talk at the 10th NASA Goddard Conference on Mass Storage Systems and Technologies, March 2002.

[83] G. A. Gibson, D. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka, *File Server Scaling with Network-Attached Secure Disks*, Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Seattle, WA, USA), ACM Press, June 1997, pp. 272–284.

[84] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, *A Cost-Effective, High-Bandwidth Storage Architecture*, Proceedings of the 8th Symposium on Architectural Support for Programming Languages and Operating Systems, ACM Press, 1998, pp. 92–103.

[85] B. Goglin and L. Prylli, *Design and Implementation of ORFA*, Tech. Report 2003-01, Laboratoire de l'Informatique du Parallèlisme, Lyon, France, September 2003.

[86] J. Gray and D. Patterson, *A Conversation with Jim Gray*, ACM Queue **1** (2003), no. 4, 8–17.

[87] J. Gray and A. Reuter, *Transaction Processing : Concepts and Techniques*, Morgan Kaufmann, 1993.

[88] K. Grimsrud and H. Smith, *Serial ATA Storage Architecture and Applications : Designing High-Performance, Cost-Effective I/O Solutions*, Intel Press, June 2003.

[89] E. Grochowski and R. D. Halem, *Technological Impact of Magnetic Hard Disk Drives on Storage Systems*, IBM Systems Journal **42** (2003), no. 2, 338–346.

[90] J. S. Hansen and R. Lachaize, *Using Idle Disks in a Cluster as a High-Performance Storage System*, in Proceedings of IEEE Cluster 2002 (Chicago, Illinois, USA), IEEE Computer Society, September 2002, pp. 245–254.

[91] J. H. Hartman, I. Murdock, and T. Spalink, *The Swarm Scalable Storage System*, Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99), IEEE Computer Society, June 1999.

[92] J. H. Hartman and J. K. Ousterhout, *The Zebra striped network file system*, ACM Transactions on Computer Systems **13** (1995), no. 3, 274–310.

[93] J. S. Heidemann and G. J. Popek, *File-system Development with Stackable Layers*, ACM Transactions on Computer Systems **12** (1994), no. 1, 58–89.

[94] ———, *Performance of Cache Coherence in Stackable Filing*, Proceedings of the 15th ACM Symposium on Operating Systems Principles, ACM Press, December 1995, pp. 110–127.

[95] Hewlett-Packard Corporation, *Configuring OPS Clusters with ServiceGuard OPS Edition*, Tech. Report B5158-90044, Hewlett-Packard, March 2002.

[96] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb, *Fast, Scalable Disk Imaging with Frisbee*, Proceedings of the 2003 USENIX Annual Technical Conference (San Antonio, TX, USA), USENIX Association, 2003, pp. 283–296.

[97] M. Hicks, J. T. Moore, and S. Nettles, *Dynamic Software Updating*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (Snowbird, Utah, USA), ACM Press, May 2001.

[98] F. Hidrobo and T. Cortes, *Autonomic Storage System Based on Automatic Learning*, Proceedings of the International Conference on High-performance Computing (HiPC 2004) (Bangalore, India), December 2004.

[99] S. Hotz, R. Van Meter, and G. Finn, *Internet Protocols for Network-Attached Peripherals*, Proceedings of the 6th NASA Goddard Conference on Mass Storage Systems and Technologies in Cooperation with 15th IEEE Symposium on Mass Storage Systems, IEEE Computer Society, March 1998.

[100] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, and R. N. Sidebotham, *Scale and Performance in a Distributed File System*, ACM Transactions on Computer Systems **6** (1988), no. 1, 51–81.

[101] H. Hsiao and D. DeWitt, *Chained Declustering : A New Availability Strategy for Multiprocessor Database Machines*, Proceedings of the 6th International Conference on Data Engineering, IEEE Computer Society, 1990, pp. 456–465.

[102] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley, *Logical vs. physical file system backup*, Proceedings of the 3rd symposium on Operating systems design and implementation, USENIX Association, 1999, pp. 239–249.

[103] N. C. Hutchinson and L. L. Peterson, *The x-Kernel : An architecture for implementing network protocols*.

[104] K. Hwang, H. Jin, E. Chow, C. Wang, and Z. Xu, *Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space*, IEEE Concurrency **7** (1999), no. 1, 60–69.

[105] K. Hwang, H. Jin, and R. Ho, *RAID-X : A New Distributed Disk Array for I/O-Centric Cluster Computing*, Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (Pittsburgh, PA), IEEE Computer Society Press, 2000, pp. 279–287.

[106] K. Hwang, H. Jin, and R. S.C. Ho, *Orthogonal Striping and Mirroring in Distributed RAID for I/O-Centric Cluster Computing*, IEEE Transactions on Parallel and Distributed Systems **13** (2002), no. 1, 26–44.

[107] IBM Almaden Research Center, *Collective Intelligent Bricks*, August 2003, See `http://www.almaden.ibm.com/StorageSystems/autonomic_storage/CIB/index.shtml` (2004-04-22).

[108] IBM Corporation, *Autonomic Computing : Creating Self-Managing Computing Systems*, See `http://www-306.ibm.com/autonomic/index.shtml/` (2004-09-06).

[109] _____, *DB2 Product Family*, See http://www-306.ibm.com/software/data/db2/ for details (2004-09-06).

[110] _____, *The Entreprise Volume Management System*, See http://evms.sourceforge.net for details (2004-05-10).

[111] IEEE, *IEEE Standard for Scalable Coherent Interface (SCI)*, IEEE, 1992, Standard 1596-1992.

[112] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li, *Improving Release-Consistent Shared Virtual Memory using Automatic Update*, Proceedings of the 2nd International Symposium on High-Performance Computer Architecture (HPCA) (San Jose, CA, USA), IEEE Computer Society, February 1996.

[113] Intel Corporation, *Hyper-Threading Technology on the Intel Xeon Processor Family for Servers*, 2002, Available at http://www.intel.com/business/bss/products/hyperthreading/server/ht_server.pdf (2004-03-02).

[114] H. Jin, G. Tan, and S. Wu, *Clustered Multimedia Servers : Architectures and Storage Systems*, Annual Review of Scalable Computing **5** (2003), 92–132, Available at http://www.comp.nus.edu.sg/~yuenck/2003/han.pdf.

[115] C. Jurgens, *Fibre Channel : A Connection to the Future*, IEEE Computer **28** (1995), no. 8, 88–90.

[116] M. Kallahalla, M. Uysal, R. Swaminathan, D. E. Lowell, M. Wray, T. Christian, N. Edwards, C. Dalton, and F. Glittler, *SoftUDC : A Software-Based Data Center for Utility Computing*, IEEE Computer **37** (2004), no. 11, 38–46.

[117] S. Kent and R. Atkinson, *Security Architecture for the Internet Protocol (IPSec)*, RFC 2401, November 1998, available at http://ietf.org/rfc.html (2004-03-01).

[118] J. O. Kephart and D. M. Chess, *The Vision of Autonomic Computing*, IEEE Computer **36** (2003), no. 1, 41–50.

[119] KernelTrap.org, *C++ in the Linux Kernel*, January 2004, Discussion available at http://kerneltrap.org/node/2067.

[120] P. B. T. Khoo and W. Y. H. Wang, *Introducing A Flexible Data Transport Protocol for Network Storage Applications*, Proceedings of the 19th IEEE Symposium on Mass Storage Systems (Adelphi, Maryland, USA), IEEE Computer Society, April 2002, pp. 241–257.

[121] K. Kim, J. S. Kim, and S. I. Jung, *GNBD/VIA : A Network Block Device over Virtual Interface Architecture on Linux* , Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS) (Fort Lauderdale, Florida, USA), IEEE Computer Society, April 15 - 19 2002.

[122] J. J. Kistler and M. Satyanarayanan, *Disconnected Operation in the Coda File System*, Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP) (Pacific Grove, CA, USA), ACM Press, 1991, pp. 213–225.

[123] A. J. Klosterman and G. Ganger, *Cuckoo : Layered clustering for NFS*, Tech. Report CMU-CS-02-183, Carnegie Mellon School of Computer Science, October 2002.

[124] B. Knowles and N. Christenson, *Design and Implementation of Highly Scalable E-mail Systems*, Proceedings of the LISA Conference (New Orleans, USA), USENIX Association, December 2000.

[125] M. Ko, J. Hufferd, M. Chadalapaka, U. Elzur, H. Shah, and P. Thaler, *iSCSI Extensions for RDMA Specification (Version 1.0)*, 2003, RDMA Consortium, available at http://www.rdmaconsortium.org/home/draft-ko-iwarp-iser-v1.PDF.

[126] D. Kotz, *Disk-directed I/O for mimd multiprocessors*, ACM Transactions on Computer Systems **15** (1997), no. 1, 41–74.

[127] N. P. Kronenberg, H. M. Levy, and W. D. Strecker, *VAXcluster : A Closely-Coupled Distributed System*, ACM Transactions on Computer Systems **4** (1986), no. 2, 130–146.

[128] R. Lachaize and J. S. Hansen, *Simplifying Administration Through Dynamic Reconfiguration in a Cooperative Cluster Storage System*, Proceedings of the 6th IEEE International Conference on Cluster Computing (Cluster 2004) (San Diego, CA, USA), IEEE Computer Society, September 2004.

[129] ———, *A Distributed Shared Buffer Space for Data-intensive Applications*, Proceedings of the 5th International Workshop on Distributed Shared Memory (DSM 2005), held in conjunction with CCGRID 2005 (Cardiff, UK), May 2005.

[130] L. Lamport, *The Part-Time Parliament*, ACM Transactions on Computer Systems **16** (1998), no. 2, 133–169.

[131] R. Latham, N. Miller, R. Ross, and P. Carns, *A Next-Generation Parallel File System for Linux Clusters*, LinuxWorld Magazine (2004), 56–59.

[132] H. Lauer and R. Needham, *On the duality of operating system structures*, Proceedings of the 2nd Internaltional Symposium on Operating Systems, IRIA, October 1978.

[133] E. K. Lee, *Highly-Available, Scalable Network Storage*, Proceedings of COMPCON '95 (San Francisco, CA, USA), IEEE Computer Society, March 1995.

[134] E. K. Lee and C. A. Thekkath, *Petal : Distributed Virtual Disks*, Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems (Cambridge, MA, USA), ACM Press, October 1996, pp. 84–92.

[135] P. Lombard and Y. Denneulin, *nfsp : A Distributed NFS Server for Clusters of Workstations*, Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002) (Fort Lauderdale, Florida, USA), IEEE Computer Society, April 2002.

[136] R. Lottiaux, B. Boissinot, P. Gallard, G. Vallée, and C. Morin, *OpenMosix, OpenSSI and Kerrighed : A Comparative Study*, Tech. Report 5399, Inria, Rennes, France, November 2004.

[137] C. Lu, G. A. Alvarez, and J. Wilkes, *Aqueduct : online data migration with performance guarantees*, Proceedings of the 1st Conference on File and Storage Technology (FAST'02) (Monterey, California, USA), USENIX Association, January 2002, pp. 219–230.

[138] C. R. Lumb, R. Golding, and G. R. Ganger, *D-SPTF : Decentralized request distribution in brick-based storage systems*, Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004) (Boston, MA, USA), ACM Press, October 2004.

[139] M. Baker (editor), *Cluster Computing White Paper*, December 2000, available at `http://dsg.port.ac.uk/~mab/Links/tfcc/WhitePaper/final-paper.pdf`.

[140] M. Mesnier, E. Thereska, D. Ellard, G. R. Ganger, M. Seltzer, *File Classification in Self-* Storage Systems"*, Proceedings of the 1st International Conference on Autonomic Computing (ICAC-04) (New York, NY, USA), May 2004.

[141] K. Magoutis, S. Addetia, A. Fedorova, M. I. Seltzer, J. S. Chase, A. J. Gallatin, R. Kisley, R. G. Wickremesinghe, and E. Gabber, *Structure and Performance of the Direct Access File System*, Proceedings of the 2002 USENIX Annual Technical Conference (Monterey, CA, USA), USENIX Association, June 2002.

[142] E. P. Markatos and G. Dramitinos, *Implementation of a reliable remote memory pager*, Proceedings of the 1996 USENIX Annual Technical Conference (San Diego, California, USA), USENIX Association, January 1996.

[143] P. Massiglia, *RAID for Entreprise Computing. A Technology White Paper*, Tech. report, Veritas Software Corporation, 2000, available at `http://eval.veritas.com/webfiles/docs/RAIDirectorWP.pdf` (2004-02-25).

[144] A. McNab, *SlashGrid - a framework for Grid aware filesystems*, March 2002, Available at `http://www.gridsite.org/slashgrid/` (2004-09-02).

[145] J. Menon, D.A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, *IBM Storage Tank - a heterogeneous scalable SAN file system*, IBM Systems Journal **42** (2003), no. 2, 250–267.

[146] M. Mesnier, G. R. Ganger, and E. Riedel, *Object-Based Storage*, IEEE Communications Magazine **41** (2003), no. 8, 84–90.

[147] Message Passing Interface Forum, *MPI-2 : Extensions to the Message-Passing Interface*, ch. 9 : I/O, July 1997.

[148] R. Van Meter, G. Finn, and S. Hotz, *VISA : Netstation's virtual Internet SCSI adapter*, Proceedings of the 8th Symposium on Architectural Support for Programming Languages and Operating Systems, ACM Press, October 1998, pp. 71–80.

[149] K. Z. Meth and J. Satran, *Design of the iSCSI Protocol*, Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03) (San Diego, California, USA), IEEE Computer Society, April 2003, pp. 116–122.

[150] J. C. Mogul, *TCP Offload Is a Dumb Idea whose Time Has Come*, 9th Workshop on Hot Topics in Operating Systems (HotOS IX) (Lihue, Hawaii, USA), USENIX Association, May 2003.

[151] C. Monia, R. Mullendore, F. Travostino, W. Jeong, and M. Edwards, *iFCP - A Protocol for Internet Fibre Channel Networking*, Tech. report, IETF, 2002, Internet-draft : draft-ietf-ips-ifcp-14.txt.

[152] J. Moore and J. Chase, *Cluster on Demand*, Tech. Report CS-2002-07, Duke University, May 2002.

[153] D. Mosberger, *Scout : A Path-based Operating System*, Ph.D. thesis, Department of Computer Science, University of Arizona, Tucson, AZ, USA, July 1997.

[154] D. Mosberger and L. Peterson, *Making Paths Explicit in the Scout Operating System*, Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96) (Seattle, WA, USA), USENIX Association, October 1996, pp. 153–168.

[155] I. Murdock and J. H. Hartman, *Swarm : A Log-Structured Storage System for Linux*, Proceedings of the USENIX Annual Technical Conference – FREENIX Track (San Diego, CA, USA), USENIX Association, June 2000.

[156] Myricom, *The GM-2 Message Passing System*, 2003, see `http://www.myri.com/scs/GM-2/doc/refman.pdf` (2004-06-10).

[157] Myricom Inc, *Sockets-GM*, See `http://www.myri.com/myrinet/performance/Sockets-GM/` for details.

[158] D. F. Nagle, G. R. Ganger, J. Butler, G. Goodson, and C. Sabol, *Network Support for Network-Attached Storage*, Proceedings of Hot Interconnects 1999 (Stanford, CA, USA), IEEE Computer Society, August 1999.

[159] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel, *Nswap : A Network Swapping Module for Linux Clusters*, Proceedings of Euro-Par'03 International Conference on Parallel and Distributed Computing, Lecture Notes in Computer Science, vol. 2790, 2003.

[160] R. Oldfield and D. Kotz, *High Performance Mass Storage and Parallel I/O*, ch. 45 : Scientific Applications using Parallel I/O, pp. 655–666, IEEE Press / John Wiley & Sons, 2001.

[161] S. W. O'Malley and L. L. Peterson, *A dynamic network architecture*, ACM Transactions on Computer Systems **10** (1992), no. 2, 110–143.

[162] Open Group Technical Standards, *Protocols for Interworking : XNFS, Version 3W*, The Open Group, February 1998.

[163] Oracle Corp, *Oracle RAC 10g Overview*, November 2003, Available at http://www.oracle.com/technology/products/database/clustering/pdf/TWP_RAC_Overview_10gR1_112503.pdf (2004-09-06).

[164] P. Lombard, *NFSP : Une Solution de Stockage Distribué pour Architectures à Grande Echelle*, Phd thesis, Institut National Polytechnique de Grenoble, Grenoble, France, December 2003.

[165] V. Pai, P. Druschel, and W. Zwaenepoel, *Flash : An efficient and portable Web server*, Proceedings of the 1999 Annual Usenix Technical Conference (Monterey, CA, USA), USENIX Association, June 1999.

[166] V. S. Pai, *IO-Lite A Copy-free UNIX I/O System*, Master's thesis, Rice University, Houston, TX, USA, 1997, Available at http://historical.ncstrl.org/tr/ps/rice_cs/TR97-269.ps.

[167] V. S. Pai, P. Druschel, and W. Zwaenepoel, *IO-Lite : A Unified I/O Buffering and Caching System*, ACM Transactions on Computer Systems **18** (2000), no. 1.

[168] Panasas Inc., *The ActiveScale File System*, 2004, see http://www.panasas.com/activescaleos.html for details (2004-06-23).

[169] D. A. Patterson, G. Gibson, and R. H. Katz, *A Case for Redundant Arrays of Inexpensive Disks (RAID)*, Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, ACM Press, 1988, pp. 109–116.

[170] B. Pawlowski, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, S. Shepler, and R. Thurlow, *The NFS Version 4 Protocol*, Proceedings of the 2nd international System Administration and Networking Conference (SANE) (Maastricht, The Netherlands), May 2000.

[171] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and David Hitz, *NFS Version 3 Design and Implementation*, Proceedings of the USENIX 1994 Summer Conference (Boston, MA, USA), USENIX Association, June 1994.

[172] G. Peng, S. Sharma, and T. Chiueh, *A case for network-centric buffer cache organization*, Proceedings of 11th Symposium on High Performance Interconnects (Hot-I 2003) (Stanford University, CA, USA), IEEE Computer Society, 2003.

[173] G. Pfister, *High Performance Mass Storage and Parallel I/O*, ch. An Introduction to the Infiniband Architecture, pp. 617–632, Wiley, 2002.

[174] M. Pillai et al., *A High Performance Redundancy Scheme for Cluster File Systems*, Proceedings of IEEE Cluster 2003 (Hong Kong), December 2003.

[175] Polyserve Inc., *Data Integrity in Cluster File Systems*, white paper, 2003, Available at http://www.polyserve.com/requestinfo_form.html (2004-09-02).

[176] K. Preslan, A. P. Barry, J. E. Brassow, G. Erickson, E. Nygaard, C. Sabol, S. Soltis, D. Teigland, and M. O'Keefe, *A 64-bit, Shared Disk File System for Linux*, Proceedings of the 7th NASA Goddard Conference on Mass Storage Systems and Technologies (San Diego, CA), IEEE Computer Society Press, March 1999, pp. 22–41.

[177] K. W. Preslan, A. Barry, J. Brassow, R. Cattelan, A. Manthei, E. Nygaard, S. Van Oort, D. Teigland, M. Tilstra, M. O'Keefe, G. Erickson, and M. Agarwal, *Implementing Journaling in a Linux Shared Disk File System*, Proceedings of the 8th NASA Goddard Conference on Mass Storage Systems and Technologies (College Park, Maryland, USA), IEEE Computer Society, March 2000.

[178] 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, and M. O'Keefe, *Scalability*

and Recovery in a Linux Cluster File System, Proceedings of the 4th Annual Linux Showcase and Conference (College Park, Maryland, USA), October 2000.

[179] L. Prylli and B. Tourancheau, *BIP : A New Protocol Designed for High Performance Networking on Myrinet*, Lecture Notes in Computer Science **1388** (1998).

[180] R. Lottiaux and C. Morin, *A cluster operating system based on software COMA memory management*, Proceedings of the 2nd Workshop on Software Distributed Shared Memory (WSDSM 2000) (Santa Fe, NM, USA), May 2000.

[181] M. Rajagopal, R. Bhagwat, and R. A. Helland, *Fibre Channel over TCP/IP (FCIP)*, Tech. report, IETF, 2002, Internet-draft : draft-ietf-ips-fcovertcpip-12.txt.

[182] R. Recio, *Server I/O Networks Past, Present, and Future*, Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence, ACM Press, 2003, pp. 163–178.

[183] Red Hat Inc., *Clusering extensions to LVM2*, See `http://sources.redhat.com/cluster/clvm/` for details (2004-09-02).

[184] G. Regnier, S. Makineni, R. Illikal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong, *TCP Onloading for Data Center Servers*, IEEE Computer **37** (2004), no. 11, 48–58.

[185] P. Reisner, *Distributed replicated block device*, in Proceedings of the 9th International Linux System Technology Conference (Cologne, Germany), September 2002, available at `http://www.drbd.org/fileadmin/drbd/publications/drbd_lk9.pdf` (2004-03-15).

[186] B. Richard, P. Augerat, N. Maillard, S. Derr, S. Martin, and C. Robert, *I-Cluster : Reaching TOP500 Performance Using Mainstream Hardware*, Tech. report, HP Labs, May 2001.

[187] E. Riedel, G. A. Gibson, and C. Faloutsos, *Active Storage for Large-Scale Data Mining and Multimedia*, Proceedings of the 24th International Conference on Very Large Data Bases (VLDB), August 1998, pp. 62–73.

[188] M. Rosenblum and J. K. Ousterhout, *The Design and Implementation of a Log-Structured File System*, ACM Transactions on Computer Systems **10** (1992), no. 1, 26–52.

[189] Y. Saito, B. Bershad, and H. M. Levy, *Manageability, Availability and Performance in Porcupine : A Highly Scalable, Cluster-based Mail Service*, Proceedings of the 17th ACM Symposium on Operating Systems Principles, ACM Press, 1999, pp. 1–15.

[190] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence, *FAB : Building Distributed Enterprise Arrays from Commodity Components*, Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004) (Boston, MA, USA), ACM Press, October 2004.

[191] K. Salem and H. Garcia-Molina, *Disk Striping*, Proceedings of the 2nd International Conference on Data Engineering, February 5-7, 1986, Los Angeles, California, USA, IEEE Computer Society, 1986, pp. 336–342.

[192] Samba Project, *Common Internet File System*, See `http://samba.org/cifs/` for details.

[193] J. Sanchez, V. Gulias, A. Valderruten, and J. Mosquera, *State of the Art and Design of VOD Systems*, in Proceedings of the International Conference on Information Systems Analysis, SCI'00-ISAS'00, July 2000.

[194] Sandia National Laboratories, *the Sandia Portals Project*, 2004, See `http://www.cs.sandia.gov/~ktpedre/portals/index.html` (2004-09-06).

[195] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, *iSCSI (internet SCSI)*, Tech. report, IETF, 2003, Internet-draft : draft-ietf-ips-iscsi-20.txt.

[196] F. B. Schmuck and R. L. Haskin, *GPFS : A Shared-Disk File System for Large Computing Clusters*, Proceedings of the 1st Conference on File and Storage Technologies (FAST), USENIX Association, 2002, pp. 231–244.

[197] M. D. Schroeder, *Computer systems : Theory, technology, and applications*, ch. 38 : Using Sharing to Simplify System Management, pp. 259–268, Springer-Verlag, New York, USA, 2004.

[198] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, *Server-Directed Collective I/O in Panda*, Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 1995.

[199] F. Seifert and H. Kohman, *A Fast Socket Implementation over SCI*, 2004, Available at `http://www.dolphinics.com/pdf/whitepapers/sci-socket.pdf`.

[200] M. Seltzer, Y. Endo, C. Small, and K. Smith, *Dealing with disaster : Surviving misbehaved kernel extensions*, Proceedings of the 2nd Symposium on Operating System Design and Implementation (OSDI) (Seattle, WA, USA), USENIX Association, October 1996.

[201] R. A. Shillner and E. W. Felten, *Simplifying Distributed File Systems Using a Shared Logical Disk*, Tech. Report TR-524-96, Princeton University, Princeton, NJ, USA, October 1996.

[202] P. Shivam and J. S. Chase, *On the Elusive Benefits of Protocol Offload*, Proceedings of the SIGCOMM Workshop on Network-I/O Convergence : Experience, Lessons, Implications (NICELI) (Karlsruhe, Germany), ACM Press, August 2003.

[203] Silicon Graphics Inc., *SGI CXFS : A High-Performance Multi-OS SAN Filesystem from SGI*, white paper, June 2004, Available at `http://www.sgi.com/pdfs/2691.pdf` (2004-09-03).

[204] SNIA Technical Council, *the Shared Storage Model*, 2003, Available at `http://www.snia.org/tech_activities/shared_storage_model/` (2004-07-08).

[205] S. Soltis, G. Erickson, K. Preslan, M. O'Keefe, and T. Ruwart, *The Design and Performance of a Shared Disk File System for IRIX*, Proceedings of the 6th NASA Goddard Conference on Mass Storage Systems and Technologies, IEEE Computer Society Press, 1997.

[206] S. Soltis, T. Ruwart, and M. O'Keefe, *The Global File System*, Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies (College Park, MD, USA), IEEE Computer Society Press, September 1996, pp. 319–342.

[207] C. Soules, J. Appavoo, K. Hui, R. W. Wisniewski, D. Da Silva, G. R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis, *System Support for Online Reconfiguration*, Proceedings of the USENIX Technical Conference (San Antonio, Texas), USENIX Association, June 2003.

[208] L. Spracklen and S. G. Abraham, *Chip Multithreading : Opportunities and Challenges*, Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11) (San Francisco, CA, USA), IEEE Computer Society, February 2005.

[209] T. Sterling, *The scientific workstation of the future may be a pile of PCs*, Communications of the ACM **39** (1996), no. 9, 11–12.

[210] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer, *BEOWULF : A parallel workstation for scientific computation*, Proceedings of the 24th International Conference on Parallel Processing (Oconomowoc, WI, USA), 1995, pp. 11–14.

[211] Storage Networking Industry Association (SNIA), *Home page of the Object-based Storage Devices (OSD) Technical Work Group*, `http://www.snia.org/tech_activities/workgroups/osd/` (2004-02-22).

[212] Sun Microsystems Inc., *XDR : External Data Representation Standard*, RFC 1014, June 1987, available at `http://ietf.org/rfc.html` (2004-03-01).

[213] ———, *RPC : Remote Procedure Call Specification Version 2*, RFC 1057, June 1988, available at `http://ietf.org/rfc.html` (2004-03-01).

[214] ———, *NFS : Network File System Specification*, RFC 1094, March 1989, available at `http://ietf.org/rfc.html` (2004-03-01).

[215] H. Sutter, *The Free Lunch is Over : A Fundamental Turn Toward Concurrency in Software*, Dr. Dobb's Journal **30** (2005), no. 3.

[216] M. Swift, M. Annamalai, B. Bershad, and H. M. Levy, *Recovering Device Drivers*, Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI) (San Francisco, CA, USA), USENIX Association, December 2004.

[217] M. Swift, B. N. Bershad, and H. M. Levy, *Improving the Reliability of Commodity Operating Systems*, ACM Transactions on Computer Systems **22** (2004), no. 4.

[218] N. Talagala, S. Asami, D. Patterson, and K. Lutz, *Tertiary Disk : Large Scale Distributed Storage*, Tech. Report UCB//CSD-98-989, University of California at Berkeley, 1999.

[219] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang, and L. Chu, *A Self-Organizing Storage Cluster for Parallel Data-Intensive Applications*, Proceedings of the 2004 High Performance Computing, Networking and Storage Conference (SC2004) (Pittsburgh PA, USA), November 2004.

[220] H. Tang and T. Yang, *An Efficient Data Location Protocol for Self-Organizing Storage Clusters*, Proceedings of the International Conference for High Performance Computing and Communications (SC2003) (Phoenix, AZ, USA), November 2003.

[221] D. Teigland and H. Mauelshagen, *Volume Managers in Linux*, Proccedings of the FREENIX Track : 2001 USENIX Annual Conference (Boston, Massachussets, USA), USENIX Association, June 2001.

[222] C. A. Thekkath, T. Mann, and E. K. Lee, *Frangipani : a Scalable Distributed File System*, Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP), ACM Press, 1997, pp. 224–237.

[223] E. Thereska, J. Schindler, J. Bucy, B. Salmon, C. R. Lumb, and G. R. Ganger, *A Framework for Building Unobtrusive Disk Maintenance Applications*, Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04) (San Francisco, CA, USA), USENIX Association, March 2004.

[224] T. Ungerer, B. Robic, and J. Silc, *A Survey of Processors with Explicit Multithreading*, ACM Computing Surveys **35** (2003), no. 1, 29–63.

[225] Veritas Corporation, *Building Robust and Highly Manageable Oracle9i Real Application Clusters with Veritas Database Edition/Advanced Cluster 3.5*, White paper, Veritas, June 2002.

[226] W. Vogels, D. Dumitriu, A. Agrawal, T. Chia, and K. Guo, *Scalability of the Microsoft Cluster Service*, Proceedings of the second USENIX Windows NT Symposium (Seattle, WA, USA), USENIX Association, August 1998.

[227] M. Volle, *Evolution du prix des micro-ordinateurs*, December 2003, Available at `http://www.volle.com/statistiques/primicro.htm` (2004-01-15).

[228] R. von Behren, E. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, and David Culler, *Ninja : A Framework for Network Services*, Proceedings of the 2002 Usenix Annual Technical Conference (Monterey, CA, USA), USENIX Association, June 2002.

[229] R. von Behren, J. Condit, and E. Brewer, *Why Events are a Bad Idea (for High-concurrency Servers)*, Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS IX) (Lihue, Hawaii), USENIX Association, May 2003.

[230] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and Eric Brewer, *Capriccio : Scalable Threads for Internet Services*, Proceedings of the 19th Symposium on Operating System Principles (SOSP-19) (Lake George, NY, USA), ACM Press, October 2003.

[231] A. Wagner, H. Jin, and D. K. Panda, *NIC-Based Offload of Dynamic User-Defined Modules for Myrinet Clusters*, Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2004) (San Diego, CA, USA), IEEE Computer Society, September 2004.

[232] M. Welsh, D. Culler, and E. Brewer, *SEDA : An Architecture for Well-Conditioned, Scalable Internet Services*, Proceedings of the 18th Symposium on Operating Systems Principles (SOSP-18) (Banff, Canada), ACM Press, October 2001.

[233] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, *An Integrated Experimental Environment for Distributed Systems and Networks*, Proceedings of the 5th Symposium on Operating Systems Design and Implementation (Boston, MA), USENIX Association, December 2002, pp. 255–270.

[234] B. White, W. Ng, and B. Hillyer, *Performance Comparison of IDE and SCSI Disks*, Tech. report, Bell Labs, 2001, Avalaible at `http://www.csl.cornell.edu/~bwhite/papers/ide_scsi.pdf` (2004-01-29).

[235] J. Wu, P. Wyckoff, D. K. Panda, and R. Ross, *Unifier : Unifying Cache Management and Communication Buffer Management for PVFS over InfiniBand*, Proceedings of IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 04) (Chicago, IL, USA), IEEE Computer Society, 2004.

[236] E. Zadok, I. Bădulescu, and A. Shender, *Extending File Systems Using Stackable Templates*, Proceedings of the Annual USENIX Technical Conference (Monterey, CA, USA), June 1999, pp. 57–70.

[237] E. Zadok and J. Nieh, *FiST : A Language for Stackable File Systems*, Proceedings of the Annual USENIX Technical Conference (San Diego, CA, USA), June 2000, pp. 55–70.

[238] Y. Zhou, L. Iftode, and K. Li, *Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems*, Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96) (Seattle, WA, USA), USENIX Association, 1996, pp. 75–88.

[239] Y. Zhu, H. Jiang, X. Qin, D. Feng, and D. R. Swanson, *Improved Read Performance in a Cost-Effective, Fault-Tolerant Parallel Virtual File System (CEFT-PVFS)*, Proceedings of the 2003 CCGRID Workshop on Parallel I/O in Cluster Computing and Computational Grids (Tokyo, Japan), IEEE Computer Society, May 2003, pp. 730–735.