Rapport de stage MIM2

Systèmes de fichiers parallèles et disques intelligents

Olivier Aumage

Stage effectué à l'Université du New Hampshire
sous la direction du professeur P. J. Hatcher

15 septembre 1998

Je tiens à remercier messieurs Philip. J. Hatcher,

Robert D. Russell et Luc Bougé ainsi que

*le contrat NSF/INRIA - C*IT.*

Quelques mots sur mon équipe d'accueil

J'ai eu la chance d'effectuer mon stage à l'Université du New Hampshire dans le charmant petit village de Durham aux Etats-Unis, au sein de l'équipe du professeur Hatcher du laboratoire d'informatique de l'UNH Les recherches de cette équipe portent essentiellement sur les systèmes de fichiers parallèles (B.P.F.S. de Robert Russell), les réseaux et protocoles de communication (Galaxy Cluster et protocole SBP.) ainsi que sur les programmes Java.

De plus, j'ai également eu l'occasion de collaborer avec Christopher Loveland du 'InterOperability Lab' (IOL) qui travaille sur un autre projet de système de fichiers parallèle fonctionnant à un niveau plus bas que BPFS. Le laboratoire IOL, installé à proximité de l'UNH, est chargé par les grands industriels du domaine des réseaux informatique de réaliser toute une panoplie de tests sur leurs produits avant commercialisation et dispose d'équipes spécialisées dans la plupart des réseaux actuels comme ATM ou *Fibre Channel*.

Introduction

Parmi les solutions répondant au besoin perpétuellement croissant de puissance engendré par le traitement de problèmes toujours plus complexes demandés aux ordinateurs, le parallélisme s'affirme sans conteste comme l'alternative la mieux appropriée. L'activité importante du monde de la recherche dans ce domaine de l'informatique en est certainement l'une des preuves les plus marquantes. Si les études effectuées dans ce secteur se sont jusqu'à présent essentiellement situées au niveau matériel pour la gestion du fonctionnement en parallèle des processeurs, et au niveau algorithmique pour la parallélisation (automatique ou non) des programmes, la nécessité d'élargir le champ de recherche aux composants logiciels et matériels périphériques afin d'optimiser leur mise en œuvre dans un environnement parallèle se fait de plus en plus sentir, afin d'accroître encore les performances en éliminant ou en élargissant les goulots d'étranglement. L'accès par les programmes parallèles aux données archivées sur disques est sans aucun doute l'un de ces thèmes à explorer du fait de son impact direct et substantiel sur les temps d'exécution.

Or l'accès à des données stockées sur disque s'effectue sur tous les systèmes actuels par l'intermédiaire d'une couche logicielle appelée 'système de fichiers', généralement partie intégrante du système d'exploitation, et l'une des caractéristiques communes à de nombreux systèmes de fichiers utilisés de nos jours est qu'ils n'ont pas été conçus pour prendre en charge de manière efficace les besoins spécifiques des programmes parallèles. Un certain nombre d'équipes de recherche se sont donc proposées de combler cette lacune en concevant divers système de fichiers dits 'parallèles', soit en réécrivant ou en modifiant la couche de gestion du système de fichier au niveau du système d'exploitation (donc au niveau 'privilégié'), soit en développant une nouvelle couche 'parallèle' servant d'interface entre le programme parallèle et le système de fichiers (mais cette fois-ci au niveau utilisateur), chacune des implémentations ayant ses avantages et ses points faibles. Le choix de R. Russell, du laboratoire d'informatique de l'Université du New Hampshire (aux Etats-Unis), avec qui j'ai travaillé au cours de mon stage, s'est porté sur la seconde option, c'est-à-dire la réalisation d'un système de fichiers parallèle appelé BPFS (pour *Basic Parallel File System*) fonctionnant au niveau utilisateur — l'argument principal étant la portabilité — dont l'absence est bien évidemment inhérente aux implémentations nécessitant une modification du système d'exploitation.

En outre, les progrès réalisés dans le domaine de la miniaturisation des équipements informatiques permettent d'envisager dans un avenir très proche la commercialisation de disques équipés de processeurs (des prototypes existent déjà), afin de leur donner une certaine 'intelligence', ou plutôt une certaine capacité de traitement par rapport à la passivité des disques actuels. Nous avons donc décidé d'étudier l'avantage qu'aurait BPFS de tirer parti de ces périphériques pour délocaliser la prise en charge des requêtes d'accès aux fichiers parallèles sur les processeurs des disques 'intelligents', et ainsi à la fois limiter les perturbations et libérer du temps de traitement sur les noeuds de calculs, avec peut-être pour conséquence une amélioration des performances.

Chapitre 1 - Présentation

Présentation du système de fichiers parallèles BPFS

Le système de fichiers parallèle BPFS — pour *Basic Parallel File System* — a été conçu et développé par R. Russell de l'Université du New Hampshire (Etats-Unis) au cours de son année passée au Laboratoire de l'Informatique du Parallélisme (LIP) à l'ENS. Les objectifs principaux qui ont guidé sa réalisation et motivé les choix technologiques sont d'une part la portabilité, une donnée clé pour assurer la pérennité d'un logiciel et pour en permettre l'étude sur une large variété d'architectures, et d'autre part la possibilité de faire fonctionner le système sur des machines et des systèmes d'exploitation non spécifiquement conçus, notamment sur du matériel courant et à faible coût.

Applications cibles

Les domaines d'application visés par le système de fichiers parallèle BPFS sont particulièrement divers, et comprennent notamment la gestion de serveurs de vidéo à la demande, le traitement d'images d'origine médicale ou envoyées par des satellites, ou encore les calculs et manipulations mathématiques sur de gros volumes de données. C'est la raison pour laquelle il a été conçu de manière à permettre aux applications clientes un accès performant à des fichiers très volumineux.

Mode d'exploitation

A l'inverse d'un certain nombre de projets actuels de systèmes de fichiers parallèles, BPFS fonctionne intégralement en mode utilisateur. Son installation ne requiert donc pas de modification du noyau du système d'exploitation, et sa mise en œuvre s'en trouve grandement facilitée car l'utilisateur désireux d'utiliser BPFS n'a pas besoin de posséder de droits privilégiés sur la machine sur laquelle il veut faire fonctionner le système de fichiers parallèle. De plus, l'absence de modifications au niveau du noyau a pour conséquence directe une portabilité accrue — ce qui est, rappelons-le, un des buts majeurs du projet — d'une part en évitant la nécessité d'une mise à jour à chaque nouvelle version du système d'exploitation sous jacent, et d'autre part en permettant l'utilisation de BPFS sur des systèmes commerciaux dont le code source propriétaire n'est généralement pas accessible, ou tout au moins pas gratuitement.

Parallélisme effectif

BPFS diffère en outre de certaines autres implémentations par le fait qu'il n'est pas dédié a une application donnée à un instant donné, mais reste au contraire en permanence disponible à toutes les applications, ce qui facilite encore son utilisation et évite un gaspillage des ressources de la machine. Et pourrait-on encore vraiment parler de système de fichiers parallèle si ses services ne pouvaient être appelés par plusieurs applications simultanément ?

Modularité

Enfin, BPFS est conçu de manière modulaire et évolutive. Seuls les protocoles entre ses différents éléments sont définis de manière stricte. L'implémentation des modules est en revanche très libre. Ceux-ci peuvent être découpés suivant des schémas différents de la distribution de base, si cela se révèle bénéfique en permettant une exploitation plus efficace de

certaines caractéristiques propres à l'architecture physique utilisée, et autorise donc l'utilisateur à introduire des optimisations le cas échéant.

Architecture du système

Nous allons maintenant étudier l'architecture proprement dite du système de fichiers parallèle BPFS et voir l'articulation de ses différents composants logiciels.

Vue d'ensemble

BPFS est constitué de trois types de modules appelés noeuds logiques sur lesquels sont répartis ses différents composants. Lorsque le système est en cours d'utilisation par une ou plusieurs applications, au moins un noeud de chaque type doit fonctionner. Les noeuds sont dits *logiques* car ils ne correspondent pas forcément à une machine particulière (plusieurs

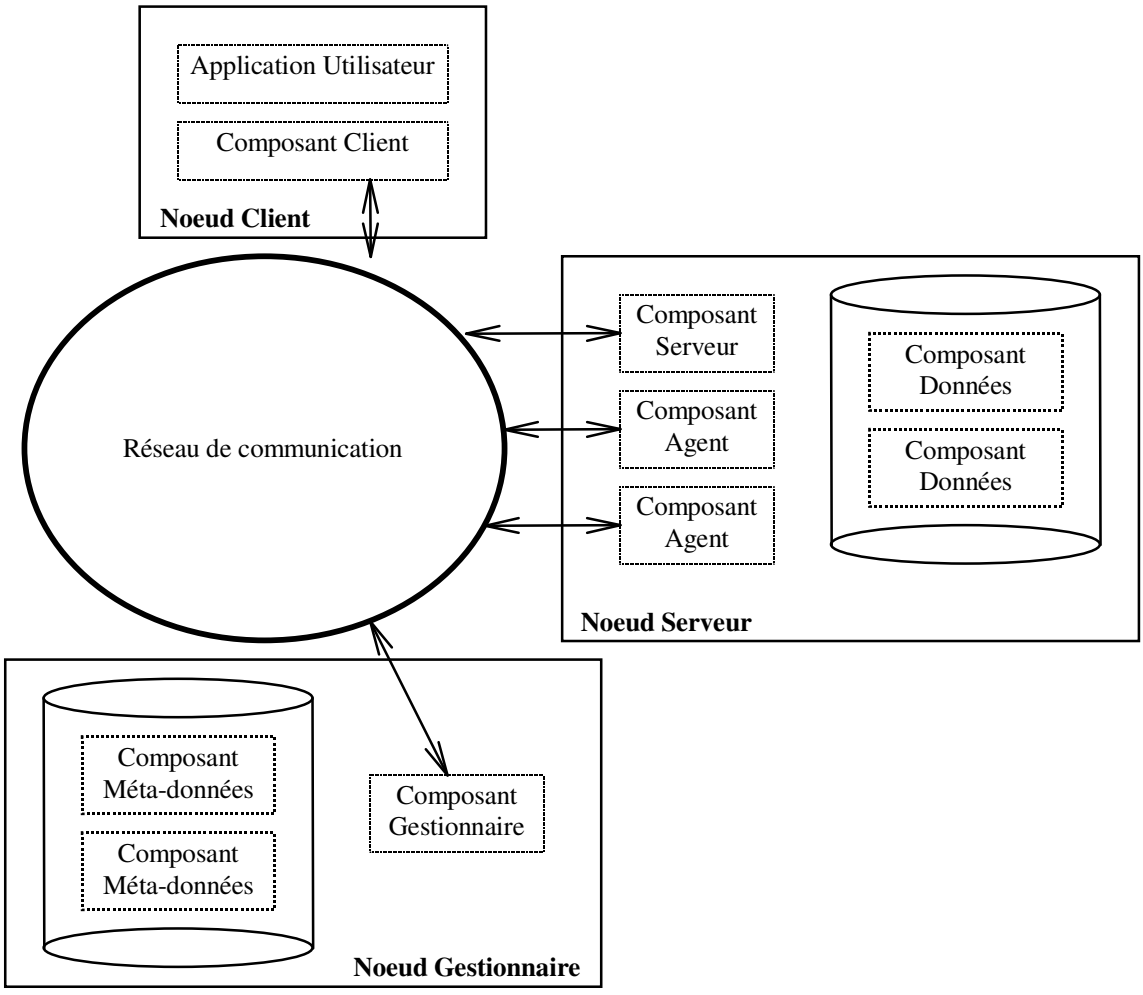

Figure 1 Noeuds et Composants de BPFS

noeuds logiques de même type ou non peuvent cohabiter sur le même noeud physique), ni nécessairement à un processus donné (le traitement en charge à un noeud particulier peut être attribué à plusieurs processus distincts). Le nombre de noeuds d'un même type n'est pas limité. Tous les noeuds communiquent entre eux à l'aide d'un réseau d'interconnexion, par l'intermédiaire d'un protocole spécifique. Le réseau peut être de type général (utilisant TCP/IP par exemple), de type spécifique (un réseau à haut débit interconnectant un groupe — ou 'cluster' — de stations de travail), ou encore une bibliothèque de communication.

Composants

Les composants de BPFS sont répartis en 2 groupes : les composants actifs et les composants passifs. Les composants passifs désignent les données gérées par le système de fichiers parallèle ainsi que les méta-données utilisées par celui-ci pour représenter l'état du système, tandis que les composants actifs regroupent les processus, 'threads' et/ou bibliothèques mis en jeu au cours d'une session BPFS.

Noeuds logiques

Les trois types de modules du système BPFS sont les suivants :

- les *noeuds clients* sont des processus de l'application utilisatrice, qui ont été liés avec le code client de BPFS (qui peut être sous forme de bibliothèque, de fichiers objets précompilés ou de fichiers sources) ;
- les *noeuds gestionnaires* (ou *manager nodes*) ont pour tâche de contrôler les données relatives aux fichiers parallèles — appelées *méta-données* — répertoriant notamment la taille et la localisation de ceux-ci sur les noeuds serveurs, et recevant les requêtes d'accès aux fichiers des applications clientes. Un fichier ne peut être géré que par un seul noeud de gestion, afin d'assurer l'intégrité des meta-données ; en revanche, il peut y avoir plusieurs noeuds de gestion opérationnels simultanément au cours d'une même session BPFS, chacun d'eux prenant à son compte la gestion d'une partie distincte de l'ensemble des fichiers parallèles utilisés par la ou les applications clientes ;
- les *noeuds serveurs* se décomposent en une partie serveur proprement dite recevant les requêtes utilisateurs interprétées et redirigées par les noeuds de gestion, et un ou plusieurs agents chargés de l'échange effectif des données avec les applications clientes, et des accès physiques aux fichiers. Un fichier peut être éventuellement être réparti sur plusieurs noeuds serveurs (technique du 'stripping' qui permet d'augmenter les performances des entrées/sorties en permettant un 'recouvrement' des accès à chaque disque), néanmoins ces derniers n'ont qu'une vue locale indépendante du reste du système (d'où la nécessité des noeuds de gestion pour centraliser les informations relatives à l'accès aux fichiers). De plus, sur un noeud serveur, un fichier n'est contrôlé que par un seul agent au plus, de manière à garantir l'intégrité des données lues ou écrites dans le fichier, de la même manière que les méta-données d'un fichier ne sont gérées que par un seul noeud de gestion. Les agents ont en outre comme responsabilité de maintenir un cache pour les fichiers qu'ils contrôlent.

BPFS utilise quatre protocoles pour régir les communications entre les types de composants susceptibles d'avoir à échanger des informations ou des données au cours d'une session :

- les communications client - gestionnaire sont constituées des requêtes provenant des applications clientes et des réponses retournées par les noeuds de gestion ;
- les communications client - agent traduisent les échangent de données depuis les fichiers vers les applications clientes et inversement ;
- les communications gestionnaire - serveur regroupent essentiellement les commandes d'ouverture et de fermeture des fichiers et de vérification des autorisations d'accès des applications clientes (ou plus précisément des utilisateurs des applications clientes), ainsi que des possibilités effectives d'accès aux fichiers modulées par les options de partages en vigueur sur ceux-ci ;
- les communications serveur - agent sont utilisées par les serveurs pour transmettre aux agents les commandes d'ouverture de fichiers ainsi que les commandes d'annulation lorsqu'un manager a été informé d'un problème d'accès sur l'un des autres noeuds serveurs

(ce qui signifie que le fichier accédé n'est pas physiquement accessible dans son intégralité et que la commande d'ouverture du fichier doit échouer).

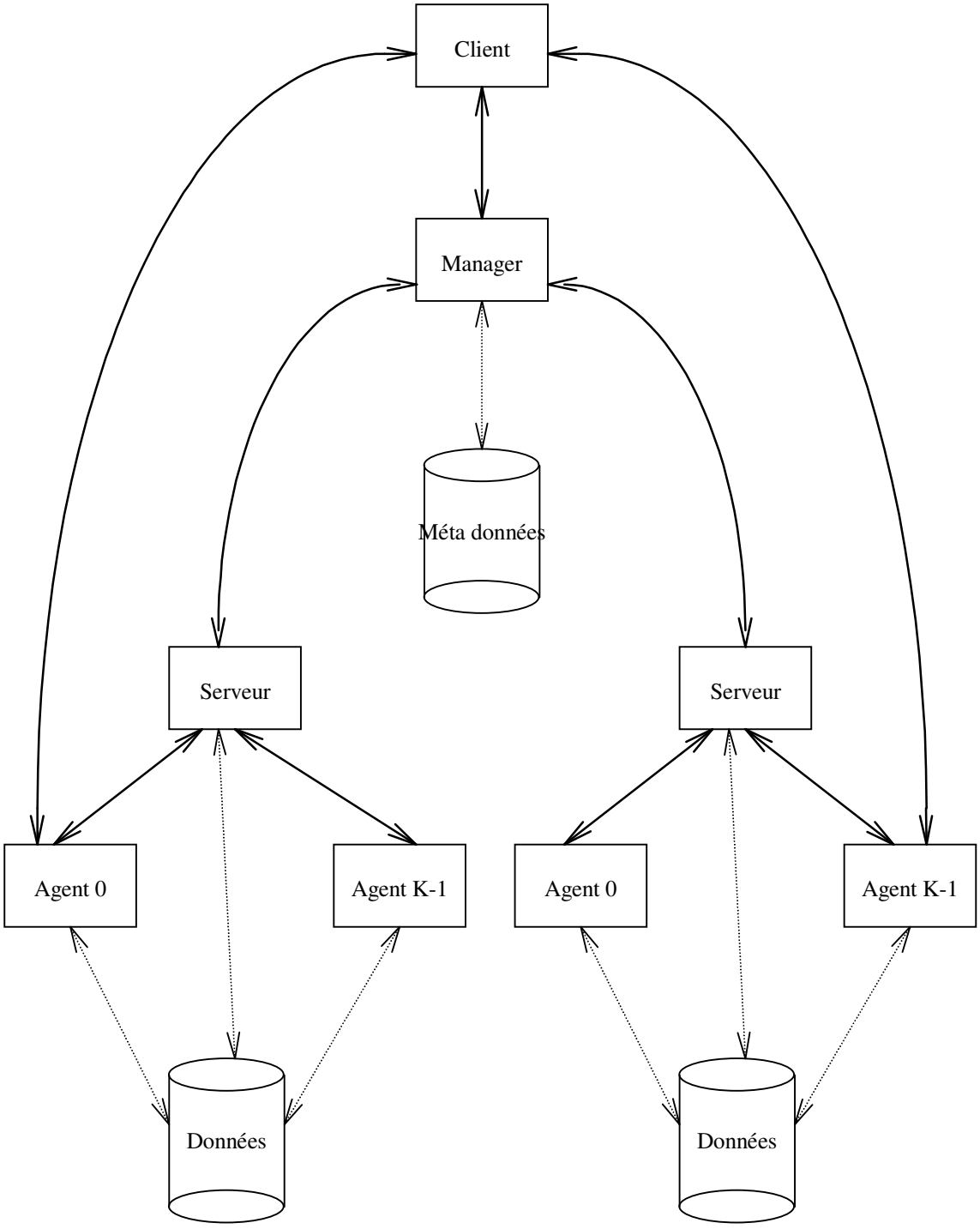

Figure 2: Vue générale de l'architecture de BPFS

Les flèches en pointillé indiquent les entrées/sorties vers des fichiers, les flèches continues indiquent les protocoles de communication.

Utilisation

Abordons à présent la manière d'utiliser BPFS dans une application cliente et les différentes méthodes disponibles pour l'accès aux fichiers, et précisons immédiatement que BPFS n'est en principe pas destiné à être utilisé seul, mais plutôt par l'intermédiaire d'une interface de programmation telle que 'API0' qui est incluse dans la distribution et qui propose un jeu de commandes directement basé sur les possibilités de BPFS, ou encore des interfaces de plus haut niveau comme MPI-IO ou une interface C mimant les commandes de la librairie standard du langage C, qui offrent plus de souplesse et une gestion des fichiers plus traditionnelle (une précision d'accès au niveau caractère au lieu du niveau bloc de BPFS).

Modes d'accès

BPFS offre 3 modes d'accès distincts aux fichiers parallèles :

- un mode 'lecture seule partagée' dans lequel seuls les accès en lecture sont autorisés (les autres accès à un fichier en lecture seule partagée doivent également être en lecture seule partagée). L'ouverture échoue si le fichier n'existe pas ;
- un mode 'écriture seule exclusive' où seule l'application ayant demandé et obtenu l'ouverture d'un fichier dans ce mode peut écrire dans le fichier. Elle ne peut en revanche pas exécuter de requêtes en lecture sur ce même fichier tant que ce mode d'accès est en vigueur. Les requêtes provenant d'autres applications sont rejetées. Une option définit le comportement de BPFS lorsque le fichier existe déjà en indiquant si une erreur doit être générée ou si le fichier doit être écrasé. De même, il est possible de demander la création du fichier lorsque celui-ci n'existe pas, au lieu de la génération d'une erreur ;
- un mode 'lecture/écriture partagées' qui permet à toute application effectuant une requête dans ce mode d'accès de mettre à jour ou de lire le fichier. Lorsqu'un fichier est en cours d'accès dans ce mode, les requêtes provenant d'autres applications et destinées au même fichiers doivent également demander un accès en lecture/écriture partagées. La consistance des accès multiples dans ce mode est préservée grâce au fait que sur un noeud serveur donné, un seul agent au plus n'a en charge la gestion d'un même fichier parallèle. Comme pour le mode lecture seule partagée, l'ouverture échoue si le fichier n'existe pas..

Ouverture d'un fichier

L'ouverture d'un fichier parallèle s'effectue de la manière décrite sur la figure 3. En voici les différentes étapes :

1) le client envoie une demande d'ouverture d'un fichier parallèle au gestionnaire en indiquant le mode d'ouverture, les options et la liste des serveurs à utiliser. Le gestionnaire vérifie alors l'existence de méta-données pour ce fichier puis retourne une erreur ou passe à l'étape suivante en créant éventuellement de nouvelles méta-données conformément aux options sélectionnées par le client ;
2) le gestionnaire transmet la requête d'ouverture aux serveurs. Les serveurs vérifient s'ils possèdent effectivement leur partie du fichier parallèle, puis ils retournent une erreur ou passent à l'étape suivante après avoir le cas échéant créé le fichier ;
3) chaque serveur crée ou réutilise un agent pour gérer la partie du fichier parallèle dont il a la charge, suivant que le fichier a été crée ou rouvert ;
4) les serveurs retournent ensuite une réponse positive au gestionnaire pour l'ouverture du fichier. Chaque serveur transmet également l'adresse de la 'socket' de connexion de son agent chargé du fichier en question ;
5) le gestionnaire transmet alors ces informations au client qui peut ainsi s'adresser aux agents à l'aide des adresses des 'sockets' de connexion ;

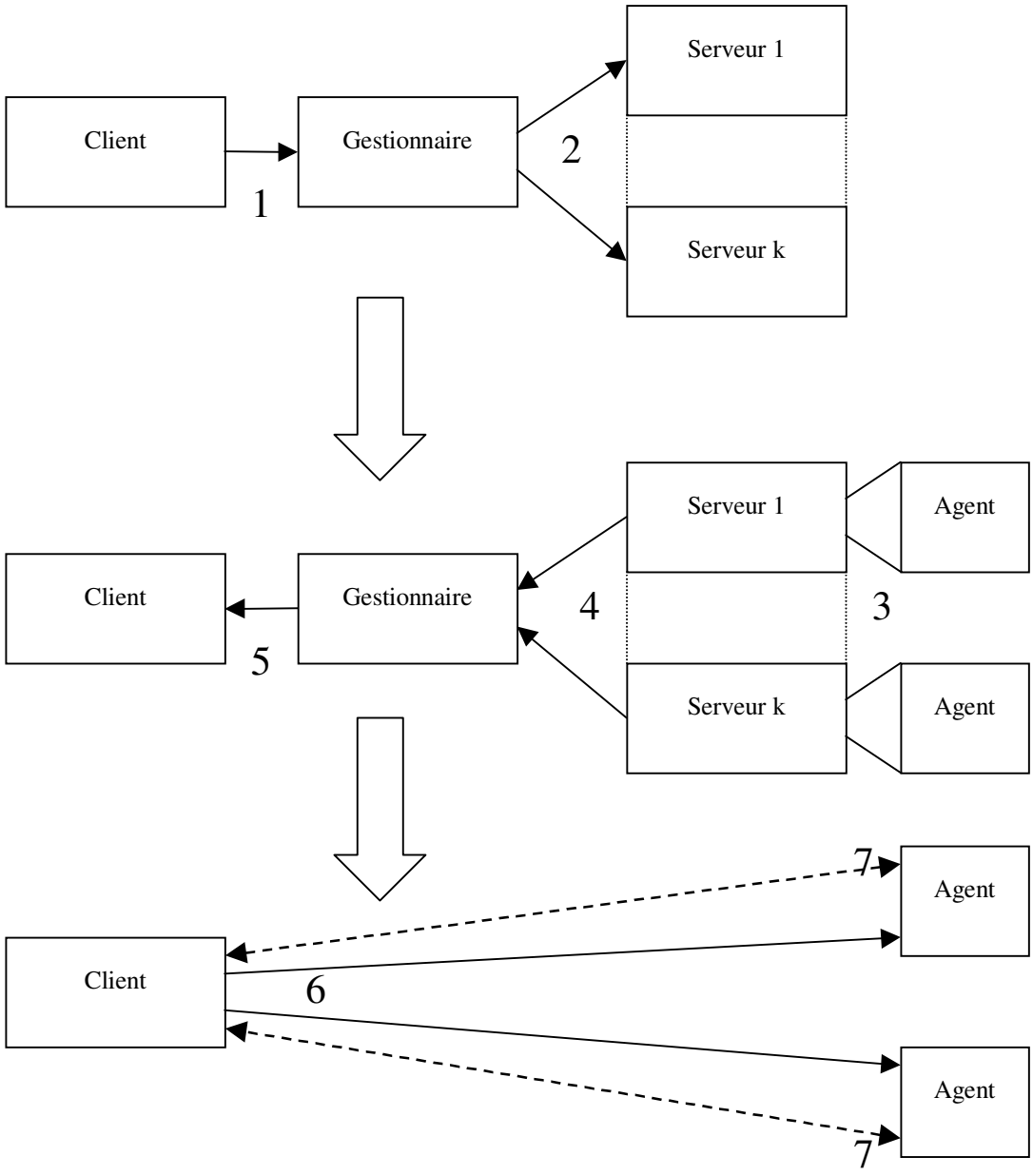

Figure 3 : ouverture d'un fichier parallèle avec BPFS.

6) le client se connecte à l'agent ;
7) l'échange de données peut alors commencer.

Modes d'utilisation

Il est possible de demander la lecture d'un ou plusieurs blocs d'un fichier parallèle de deux manières différentes. La méthode la plus simple est la lecture bloc par bloc. Elle est aussi celle qui présente le moins d'intérêt. En effet, avec ce type de lecture, les blocs sont demandés un par un séquentiellement, ce qui est d'un usage limité en programmation parallèle. Chaque commande d'accès à un bloc du fichier (en lecture ou en écriture) est immédiatement suivie d'une commande associée chargée d'attendre la terminaison de la procédure. Le seul parallélisme possible est l'exécution de calculs entre la commande d'exécution et la commande d'attente pour recouvrir les entrées/sorties par du calcul.

Afin de tirer parti au mieux des possibilités de BPFS, la seconde méthode d'utilisation est fortement recommandée. Il s'agit du mode 'flot' (stream mode). Dans ce mode, il est possible de spécifier un bloc de départ, un certain nombre de blocs à lire (ou bien lire le fichier jusqu'à la fin), et éventuellement une vitesse de transfert. Les blocs sont alors automatiquement lus sans intervention de l'application cliente, au taux de transfert demandé ou le plus vite possible si le taux de transfert n'a pas été défini, et envoyés à l'application. En utilisant ce mode, l'obtention effective de parallélisme au niveau des entrées/sorties est réalisable, car il est parfaitement possible d'établir des flots de lecture avec des noeuds serveurs distincts, qui enverront les données provenant des fichiers de façon concurrente. Avec toujours la possibilité d'exécuter des calculs en parallèle avec les accès aux fichiers.

Chapitre 2 - Réalisation

Dans la version originale de BPFS, les noeuds serveurs sont destinés à être exécutés sur des noeuds physiques à part entière, ce qui signifie qu'au mieux, les machines sur lesquelles fonctionnent les noeuds serveurs doivent traiter les requêtes d'accès au fichiers parallèles en plus de leur calculs, et au pire, des machines peuvent être dédiées uniquement au service de fichiers et donc n'effectuer aucun calcul.

Or, des disques dits 'intelligents' — des disques disposant d'un processeur, d'une mémoire, d'une connexion à un réseau, et éventuellement d'autres fonctionnalités plus ou moins élaborées comme l'encryption matérielle des communications — ont été conçus récemment. Utiliser ce type de disques comme hôtes pour les noeuds serveurs de BPFS pourrait donc être une bonne chose dans la mesure où cela permettrait de libérer complètement les noeuds de calculs des tâches de service, avec comme conséquence une diminution des perturbations à la fois sur les noeuds de calculs et sur les noeuds de service, et peut-être une amélioration sensibles des performances.

En vue d'utiliser à terme ce type de périphérique pour mettre en œuvre BPFS, je me suis proposé d'étudier d'une part les modifications à apporter au système de fichiers parallèle pour l'adapter à ce que pourrait être le système d'exploitation sur les disques *intelligents* et d'autre part l'impact sur les performances qu'aurait l'utilisation de ces disques pour les noeuds serveurs.

Toutefois, apporter des modifications à un système tel que BPFS, totalisant plus de 7000 lignes de code source en langage C hautement optimisé et faisant appel à un large éventail de concepts tels que les sockets, le protocole TCP/IP, les tubes de communication interprocessus, ainsi qu'à toute la panoplie de fonctions de gestion de fichiers demande une connaissance parfaite de son fonctionnement interne sans quoi le risque d'introduire des erreurs serait beaucoup trop important et mettrait en péril la stabilité du système. L'une des conséquence serait sans aucun doute la perte de données ce qui n'est pas admissible pour ce type d'application. J'ai donc dû, avant toute chose, me familiariser avec le code de BPFS, ce qui s'est ensuite traduit concrètement par une phase de débogage considérablement facilitée, une donnée toujours appréciable.

Mais revenons tout d'abord à la notion de disques intelligents.

Disques intelligents

Un disque 'intelligent' est un disque qui dispose d'une capacité locale de traitement d'information, c'est-à-dire un microprocesseur et une mémoire vive permettant d'exécuter du code de programme, et donc de réaliser des opérations localement. Ces disques sont connectés directement sur le réseau reliant les machines qui les utilisent à l'aide d'une interface de communication elle aussi intégrée au disque intelligent.

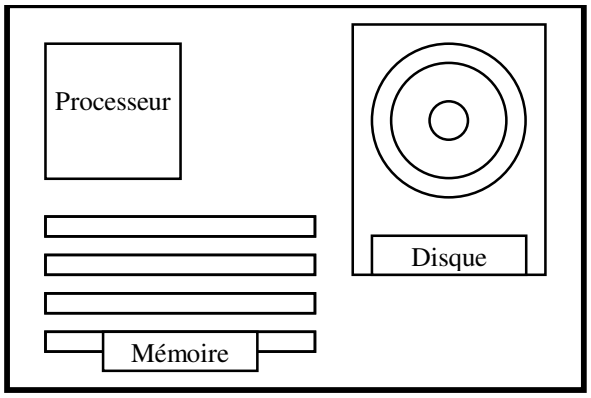

Figure 4 : un disque 'intelligent'.

Caractéristiques

Ne disposant pas de tels périphériques au laboratoire où j'ai effectué mon stage (ceux-ci sont très récents), j'ai dû déterminer quelles caractéristiques 'raisonnables' devrait posséder le système d'exploitation d'un disque intelligent pour faire fonctionner BPFS, puis adapter BPFS à l'absence des fonctionnalités non strictement indispensables.

BPFS ayant été écrit pour être portable, il est très peu exigeant quant au système sous-jacent qu'il nécessite, et peut virtuellement fonctionner sur tout système disposant d'un système de fichier et d'un réseau de communication avec un minimum de modifications (adaptation aux interfaces de programmation). Or un disque intelligent est conçu pour être connecté sur un réseau donc on peut supposer qu'il dispose d'une interface pour accéder à ce réseau. On peut également supposer que le disque dispose d'un système de fichier — même rudimentaire — car l'accès aux fichiers est réalisé par le disque.

Les caractéristiques indispensables au fonctionnement de BPFS sont donc présentes dans un système utilisant des disques intelligents. En revanche, les noeuds serveurs de la version originale de BPFS s'appuient sur un système d'exploitation multitâche car les processus serveurs créent des processus agents pour chaque fichier ouvert par les applications clientes. Les processeurs des disques intelligents n'étant en principe pas destinés à effectuer de gros calculs ou à faire fonctionner des applications complètes, on peut difficilement supposer sans hésitation que ceux-ci soient dotés de fonctionnalités avancées comme la gestion de la commutation de tâches, et même dans le cas où le multitâche serait supporté par le processeur, il n'est pas non plus certain qu'il le soit par le système d'exploitation, car il n'est pas fondamental pour les programmes susceptibles de fonctionner sur un disque intelligent. J'ai donc dû modifier la partie 'serveur' de BPFS afin qu'elle puisse fonctionner dans un environnement monotâche.

Un serveur mono-processus

La principale difficulté dans la conversion d'un programme multi-processus en un programme mono-processus est de supprimer les optimisations qui ont été introduites pour tirer parti de l'utilisation de plusieurs processus pour traiter une série de requêtes. En effet, dans le cas de la version originale de BPFS, lorsqu'une application cliente envoie une requête à un agent d'un noeud serveur, le fichier à lire ou à écrire est désigné implicitement car il y un agent par fichier et un fichier par agent. Or nous voulons effectuer le traitement de toutes les requêtes avec un seul processus ayant la double qualité de serveur et d'agent. De plus, nous ne pouvons pas nous permettre de fermer et d'ouvrir un nouveau fichier à chaque nouvelle requête demandant

un accès à un fichier différent, car cela rendrait inutile le cache de BPFS. Il faut donc que l'agent puisse désormais ouvrir simultanément plusieurs fichiers. La propriété 'un seul fichier par agent' n'est alors plus vraie et les fichiers ne sont plus désignés implicitement. Il en résulte qu'il faut donc utiliser un autre moyen d'identifier les fichiers dans les requêtes et modifier le fonctionnement du programme pour indexer les données relatives à l'accès aux fichiers par rapport à cet identifiant.

Lorsqu'un fichier est ouvert sur un noeud serveur, le processus serveur lui attribue un numéro unique qui sert à repérer le fichier concerné par les messages échangés dans les protocoles de communication de BPFS. Dans l'implémentation initiale, ce numéro n'est pas utilisé autrement qu'à des fins de débogage, en permettant de réaliser une trace des messages. Cependant, le numéro étant unique au niveau d'un serveur, il constitue un identifiant idéal pour l'usage que nous souhaitons en faire. En effet, d'une part, cela nous évite de rajouter un champ aux messages de BPFS, ce qui permet de préserver la compatibilité des protocoles de communication et surtout de ne pas nécessiter de modification des autres composants de BPFS (noeuds managers et 'souche' cliente dans les applications utilisant BPFS). Cette transparence des modifications vis-à-vis du reste du système permet en outre d'utiliser les deux types de noeuds serveurs (mono et multi-processus) au sein d'une même session BPFS. Il en résulte une souplesse d'utilisation accrue, notamment dans un environnement hétérogène.

Structures de données

L'utilisation de l'indexation des messages par un numéro de fichier se traduit dans le code source du serveur par l'ajout d'un tableau de pointeurs sur des structures regroupant les informations nécessaires au contrôle des fichiers : la suppression de l'utilisation du traitement multiprocessus est compensée par l'ajout d'une dimension aux structures de données du serveur. La taille de cette dimension détermine le nombre maximal de fichiers pouvant être ouverts simultanément.

Modifications du serveur

Le premier ajout au code source du serveur est l'ouverture d'une seconde 'socket' d'écoute destinée à recevoir les demandes de connexion des processus clients vers les 'ex-agents' (la première socket d'écoute reçoit les requêtes de connexion des processus managers au serveur).

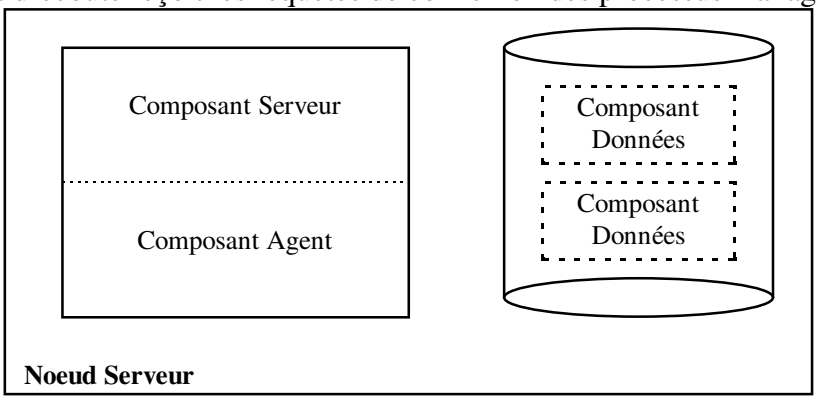

Figure 5 : composant serveur et composant agent en un seul processus.

Cet ajout implique le déplacement des routines d'acceptation de connexion et de traitement des requêtes depuis l'agent vers le nouveau serveur. Un problème se pose alors. En effet, les informations relatives à une connexion nouvellement créée doivent être enregistrées en mémoire et pouvoir ensuite être retrouvées. Or nous ne pouvons utiliser l'index du fichier car

au moment de la connexion celui-ci n'est pas encore connu (le numéro du fichier est indiqué dans tous les messages, mais la connexion intervient *avant* l'envoi du premier message.). On pourrait penser à retarder la création de la structure jusqu'à la réception du premier message, mais rien ne permet de dire quand ce message arrivera et d'autres connexions peuvent être demandées avant que celui-ci n'arrive ce qui deviendrait délicat à traiter convenablement. Une solution simple est de ne pas indexer les structures relatives aux connexions par le numéro de fichier mais par le descripteur de la socket affectée à chaque connexion (qui lui est connu dès la connexion). La portion de code ci-dessous, extraite de la partie 'déclaration de variable' du nouveau processus serveur, montre l'indexation des données par le descripteur de socket.

```
/* holds stride for each streaming client, else 0 -- indexed by client fd */
static unsigned int *streaming = NULL;

/* holds ptr to stream rate for each client -- indexed by client fd */
static struct timeval **streamrate = NULL;

/* holds ptr to stream clock for each client -- indexed by client fd */
static struct timeval **streamclock = NULL;

/* holds location structure for each connected client -- indexed by client fd */
static struct q_location *client_addr = NULL;

/* holds file record pointer for each connected client -- indexed by client fd */
static struct file_record **client_file = NULL ;
```

Il faut ensuite ajouter un champ à la structure de donnée de connexion contenant le numéro du fichier utilisé par la connexion (et défini à la valeur -1 avant que ce numéro ne soit reçu). Cette méthode permet de maintenir l'accès simultané par plusieurs clients à un même fichier.

Il reste encore à finaliser le transfert du code 'agent' dans le fichier source du serveur monoprocessus. La difficulté principale est de remplacer les communications entre les agents et le serveur (qui ont lieu par l'intermédiaire de tubes car il y a une relation de parenté entre le serveur/père et les agents/fils) par du code en ligne car ces communications peuvent donner lieu à plusieurs échanges d'informations en cascade qu'il faut alors réduire à l'aide d'imbrications de *if* et de *else* (parfois jusqu'à 5 ou 6 niveaux), tout en prenant soin d'appliquer toutes les optimisations pouvant résulter de l'expansion en ligne du code source, un peu à la manière d'un compilateur optimisant.

Nous obtenons alors — après environ trois semaines de programmation et une semaine de débogage et de vérification du bon fonctionnement du système — un processus serveur/agent avoisinant les 1700 lignes de code source C et capable de fonctionner sur des processeurs très 'rudimentaires' et ne disposant pas de mécanisme de pagination comme l'Intel 8086 et surtout le Motorola 68000 plus souple d'utilisation et très largement utilisé pour des applications très variées, bien qu'une architecture 32 bits — ou mieux 64 bits — soit fortement recommandée pour obtenir des performances acceptables car BPFS a avant tout été conçu pour être le plus efficace possible sur ce type de machines. De plus, le fait que le serveur fonctionne sous la forme d'un seul processus offre la possibilité d'utiliser un système d'exploitation allégé de tout la gestion de commutation de tâches et disposant de beaucoup moins de fonctionnalités qu'un systèmes d'exploitation classique : peu ou pas de communication interprocessus excepté la gestion du réseau, pas de gestion de mémoire, pas de mémoire virtuelle.

Nous pouvons à présent aborder l'étape d'évaluation des performances afin d'étudier le comportement de BPFS et le gain potentiel de notre serveur capable de fonctionner sur un disque 'intelligent'.

Chapitre 3 - Mise en œuvre

Comme il était possible de le prévoir, les deux versions de noeuds serveurs, l'un fonctionnant en environnement multitâche et l'autre en environnement monotâche, présentent sensiblement les mêmes performances lors de tests sur des systèmes identiques, avec un très léger avantage pour la version multitâche, ce qui était également prévisible. Nous ne nous attarderons donc pas sur la comparaison entre les deux noeuds serveurs mais nous nous attacherons plus particulièrement d'une part à déterminer l'influence des différents composants physiques du système sur les performances, et d'autre part à mesurer une éventuelle différence de performances entre une session utilisant des serveurs multitâches sur des machines très chargées et une session de référence utilisant des serveurs monotâches sur des machines non chargées (cas typique de l'utilisation sur un disque intelligent) afin de mesurer le gain de performance que l'on peut espérer obtenir.

Influence des composants d'un système

Environnement de test

Tous les tests ont été effectués sur le '*Galaxy Cluster*' du laboratoire d'informatique de l'Université du New Hampshire, qui est composé de 8 machines identiques de type PC reliées par deux réseaux de type Ethernet (un réseau fonctionnant à 10 Mbits/s et un réseau fonctionnant à 100 Mbits/s) et possédant les caractéristiques techniques suivantes :

- processeur : Pentium 133 Mhz ;
- mémoire : 32 Mo ;
- disque : de type SCSI, vitesse de transfert en lecture et en écriture : environ 6 Mo/s.

Les tests qui suivent ont été réalisés en utilisant le réseau Ethernet 100 Mbit/s. Sauf mention contraire, les machines étaient connectées par un 'switch' lors de ces tests, nous verrons plus loin que ce détail à une importance non négligeable.

Performances pures du réseau

Afin de déterminer une base de travail, j'ai tout d'abord mesuré la vitesse maximale qu'il était effectivement possible d'atteindre sur le réseau en effectuant un simple test d'envoi massif de données entre 2 machines.

Taille en octets	64000000	128000000	256000000	512000000
Temps moyen (s)	10.0	19.4	39.0	79.5

Tableau 1 : performances brutes du réseau.

La vitesse de transfert du réseau est donc d'environ 6.4 Mo/s à comparer aux 12.5 Mo/s théoriques (100 Mbits/s = 12.5 Mo/s) — la pile TCP/IP réduit donc de moitié la bande passante du réseau — et aux 6 Mo/s des disques SCSI ce qui signifie que la bande passante du réseau et des disques sont du même ordre de grandeur.

Maintenant que nous connaissons la vitesse maximale que nous pouvons obtenir sur le réseau, nous allons mesurer la vitesse réelle atteinte en utilisant BPFS en fonction du nombre de noeuds serveurs.

Performances des communications avec BPFS

Au cours des tests suivants, les entrées/sorties ont été désactivées afin de ne mesurer que les performances des communications. Chaque session contient, en plus des serveurs, un noeud de gestion et un noeud client. Les mesures de débit en lecture ont été réalisées en utilisant le mode 'flot' de BPFS.

	Débit en écriture (Mo/s)	Débit en lecture (Mo/s)
1 serveur	4.3	6.4
2 serveurs	4.3	6
3 serveurs	4.3	6
4 serveurs	4.25	4
5 serveurs	4.3	3
6 serveurs	4.3	2.7
7 serveurs	/	2.4

Tableau 2 : performances des communications (connexion par switch)

On constate tout d'abord que le débit en écriture est approximativement constant, ce qui s'explique par le fait que pendant l'écriture, le client envoie les données à chaque serveur l'un après l'autre. Il n'y a donc qu'une communication sur le réseau à un instant donné, entre le client et l'un des serveurs. Il en résulte une absence de congestion au niveau du switch, et le nombre de serveurs n'a donc pas d'influence sur le débit. En revanche, en ce qui concerne la lecture, ce paramètre a une importance beaucoup plus grande. En effet, si la vitesse de transfert n'est quasiment pas dégradée jusqu'à 3 serveurs, elle décroît ensuite sensiblement. La raison principale de ce phénomène est que contrairement à ce qui se passe dans la phase d'écriture, dans la phase de lecture, la grande majorité des communications a lieu dans le sens des serveurs vers le client. Il se crée donc une congestion au niveau du switch qui devient alors le goulot d'étranglement du système, et les performances chutent. L'absence de mesure en écriture pour le tests utilisant 7 serveurs est due à un comportement particulièrement étrange du système — dont la cause reste encore à déterminer faute de temps — qui a pour conséquence une chute brutale du débit à un niveau qui n'est plus représentatif du tout, si ce n'est de la présence d'un problème. Celui-ci n'est apparemment pas causé par le mauvais fonctionnement d'une des machines. L'apparition de ce phénomène est d'autant plus surprenante qu'en phase d'écriture, le nombre de serveurs n'a logiquement aucune influence sur le débit (ce qui est vérifié pour les tests utilisant de 1 à 6 serveurs).

Pour en revenir aux tests de lecture, la décroissance importante du débit lorsque le nombre de serveurs devient supérieur à 3 m'a incité à réaliser la même série de tests en utilisant un 'hub' à la place du 'switch' pour connecter les machines. Pour mémoire, un hub simule un réseau reliant *toutes* les machines alors qu'un switch commute une connexion entre des paires de machines. L'une des différences que cela implique réside dans le traitement des congestions, d'où l'intérêt éventuel de cette seconde mesure.

	Débit en écriture (Mo/s)	Débit en lecture (Mo/s)
1 serveur	4.4	6.0
2 serveurs	4.5	5.9
3 serveurs	4.5	5.9
4 serveurs	4.5	5.8
5 serveurs	4.5	5.8
6 serveurs	4.4	5.8
7 serveurs	/	5.8

Tableau 3 : performances des communications (connexion par hub)

A la lecture de cette seconde série de mesures, nous pouvons remarquer des différences parfois importantes avec la série réalisée en utilisant le switch, et retenir trois points importants :

- le débit en écriture est légèrement supérieur avec le hub. Ceci s'explique par le fait que le hub est essentiellement passif alors que le switch est un composant actif ce qui se traduit par l'introduction d'une latence lors de l'envoi des messages de communication ;
- le débit en lecture décroît nettement moins vite en fonction du nombre de serveurs lorsque le hub est utilisé. C'est ici qu'intervient la différence dans le traitement des congestions des deux périphériques. En effet, le hub réalisant une sorte de 'câble Ethernet virtuel' auquel seraient reliées toutes les machines, la gestion de la congestion est réalisée au niveau du protocole Ethernet lui même : lorsqu'une collision est détectée (deux messages ou plus ont été envoyés sur le support de communication), la transmission est annulée, et chacun des messages impliqués dans la collision est retransmis au bout d'un temps choisi aléatoirement par la carte Ethernet qui l'a émis. La congestion est donc traitée au niveau le plus bas, et dès que le problème est détecté, d'où une efficacité maximale. Ce mécanisme n'est cependant pas réalisable avec un switch car celui-ci isole toutes les communications. Il ne peut donc jamais y avoir une collision de deux messages au niveau du support de communication, et la gestion de collision du protocole Ethernet ne peut pas agir en cas de congestion. La conséquence de cette caractéristique est que lorsque la capacité des tampons du switch est dépassée, celui-ci détruit purement et simplement tout nouveau message arrivant, sans toutefois en avertir sa carte de communication émettrice. Celle-ci ne détecte le problème que lorsque le protocole TCP/IP lui signale que le temps d'attente du message est écoulé et que celui-ci doit être retransmis, ce qui est autrement plus coûteux que la gestion de collision par Ethernet. D'où la différence de performance observée entre les deux séries de mesures ;
- le problème rencontré lors de la phase d'écriture avec 7 serveurs dans la première série de mesures est encore présent dans la seconde série de mesures.

Ces deux séries de tests nous ont permis d'avoir une idée plus précise de la part des communications dans les performances globales du système (rappelons que ces deux tests ont été effectués sans les entrées/sorties) ainsi que de l'influence des composants du réseau. Il nous reste maintenant à aborder l'étude du gain apporté par l'utilisation des serveurs de BPFS sur des disques intelligents (simulés par des machines n'effectuant pas de calculs) par rapport à l'utilisation des serveurs sur des machines de calcul (avec une charge de calcul importante).

Gain de performances

Les tests suivants ont été réalisés avec des sessions comprenants 4 noeuds serveurs, 1 noeud de gestion et un noeud client, avec les entrées/sorties activées. Le réseau est connecté au switch. Le volume de données lues est de 10000 blocs de 16 Ko, soit environ 40 Mo sur chacun des quatre serveurs. Les tests utilisent le mode 'flot'. Les machines chargées exécutent 4 processus produisant une charge de calcul d'environ 1 chacun (soit une charge approximative de 4).

Mesure	Temps	Débits
Machine non chargée	41.54 s	3.8 Mo/s
Machine chargée	43.9 s	3.55 Mo/s

Tableau 4 : performances sur des serveurs chargés et sur des serveurs non chargés.

On peut tout d'abord remarquer que la différence en temps d'exécution est relativement faible, et que la diminution du débit sur une machine chargée n'est pas très élevée non plus, bien que non négligeable (la décroissance du débit est d'environ 7%). Il faut d'une part noter que le fichier utilisé, d'une taille de près de 160 Mo, est relativement petit par rapport aux champs d'application visés par BPFS. Sur des données de plusieurs gigaoctets, voire plusieurs téraoctets pour des bases de données vidéo, une différence de 250 Ko/s n'est plus négligeable. De plus, comme nous l'avons noté, le débit du réseau devient inférieur à celui des disques SCSI dès que l'on utilise plus de 3 serveurs (sur le système utilisé pour les tests). Il en résulte que la majorité du temps de traitement est utilisée dans les communications. Il serait probablement très intéressant de reproduire ces tests en utilisant un réseau plus rapide comme le Myrinet de la machine PoPC du LIP. En effet, sur une telle machine les données changent radicalement : la bande passante du réseau est d'environ 1 Gbits/s alors que celle des disques — toujours du type SCSI — reste inchangée. Le débit du réseau devient alors très supérieur à celui des disques, et le goulot d'étranglement du système s'en trouve déplacé depuis les communications vers les entrées sorties. Nul doute que la charge de la machine aurait alors une incidence beaucoup plus grande sur les temps d'exécution mesurés.

Conclusion

Au cours de mon stage de seconde année du Magistère d'Informatique j'ai donc étudié la possibilité d'utiliser le système de fichiers BPFS (*Basic Parallel File System*) de R. Russell sur une architecture qui ferait appel à des disques dit '*intelligents*' et dotés d'une mémoire ainsi que d'un processeur. J'ai tout d'abord proposé un ensemble de caractéristiques minimales que devraient posséder de tels disques afin qu'ils puissent faire fonctionner les processus serveurs du système de fichiers parallèle et j'ai ensuite modifié ces processus serveurs afin de les rendre compatibles avec ces caractéristiques minimales tout en maintenant la compatibilité avec les autres modules de BPFS ainsi qu'avec les différents protocoles de communication inter-modules. Enfin j'ai réalisé un certain nombre d'expériences afin de déterminer dans quelles conditions une amélioration des performances pourrait être obtenue par l'utilisation de BPFS sur un système à base de disques intelligents à la place d'un système classique, et de quel ordre de grandeur serait le gain de performance. Ces tests ont mis en évidence la nécessité de disposer d'un réseau d'interconnexion très rapide et d'éléments de connexion efficace avant d'espérer obtenir une augmentation substantielle des performances. De plus, j'ai eu l'occasion de débuter une collaboration avec C. Loveland de l'InterOperability Lab à l'UNH sur un autre projet — très difficile techniquement — de système de fichiers parallèles fonctionnant au niveau du noyau de Linux, ce qui m'a permis de découvrir l'architecture interne remarquable de la gestion des fichiers sur ce système d'exploitation.

Le nombre et l'importance des différents paramètres entrant en jeu dans les temps d'exécution mesurés au cours de mes expérimentations avec BPFS est tel qu'il aurait été particulièrement intéressant de pouvoir réaliser des tests sur des architectures très variées, et notamment des architectures disposant de réseaux à haut débit. L'idéal aurait probablement été de pouvoir essayer BPFS sur de véritables disques intelligents car toute simulation a ses limites et sa représentativité du cas réel possède toujours une part de subjectivité dont il faut se méfier. Enfin, il aurait probablement été utile de pouvoir mener une investigation poussée en ce qui concerne les phénomènes étranges relevés dans certaines configurations d'utilisation de BPFS.

Toutefois, il ne s'agit que d'une étude préliminaire et l'évolution rapide des performances des composants informatiques permet d'envisager un intérêt croissant des disques intelligents comme hôtes des serveurs de BPFS, notamment grâce aux progrès considérables réalisés dans le domaine des réseaux, car on pourra alors observer un véritable parallélisme lors de la lecture de fichiers parallèles, grâce à un recouvrement des entrées sorties de chaque disque. De plus, d'autres aspects restent à aborder en vue de proposer un système véritablement finalisé. L'un d'eux — et non des moindres — concerne d'une part la sécurité des communications entre les différents composants et d'autre part l'authentification et le contrôle de la validité des requêtes envoyées aux composants agents qui ne disposent que d'une vue locale du système. Or certains prototypes de disques intelligent intègrent des circuits de cryptage et de décryptage de données plus rapides et surtout plus sûrs que l'exécution d'algorithmes équivalents par des processeurs polyvalents. Ce dernier point pourrait donc constituer un avantage important (voire déterminant pour certaines applications pointues) en faveur de l'utilisation de disques intelligents.

Bibliographie

The Architecture of BPFS : a Basic Parallel File System Version 1.0
Robert D. Russell, L.I.P. - rapport de recherche n° 98-21, Mars 1998.

Application Interfaces to BPFS : a Basic Parallel File System
Robert D. Russell, L.I.P. - rapport de recherche n° 98-28, Juin 1998.

A Study of End-User SCSI Disk Performance with Parallel and Non-parallel Files
Robert D. Russell, L.I.P. - rapport de recherche n° 98-38, Août 1998.

Network-Attached Secure Disks
URL : **`HTTP://www.pdl.cs.cmu.edu/NASD`**
Carnegie Mellon University.