N° d'ordre: 2619

THÈSE

Présentée

devant l'Université de Rennes 1

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITÉ DE RENNES 1
Mention INFORMATIQUE

par

Renaud LOTTIAUX

Équipe d'accueil : IRISA/PARIS
École Doctorale : Mathématiques, Informatique, Signal et Électronique et Télécommunications
Composante universitaire : IFSIC

Titre de la thèse :
Gestion globale de la mémoire physique d'une grappe pour un système à image unique : mise en œuvre dans le système GOBELINS

soutenue le 18 décembre 2001 devant la commission d'Examen

Mme.	Françoise	ANDRÉ	Président
MM.	Xavier	ROUSSET DE PINA	Rapporteurs
	Willy	ZWAENEPOEL	
MM.	Franck	CAPPELLO	Examinateurs
	Thierry	PRIOL	
Mme.	Christine	MORIN	

Remerciements

Je remercie vivement Françoise André, professeur à l'Université de Rennes 1, qui me fait l'honneur de présider ce jury.

Je tiens également à remercier ma directrice de thèse Christine Morin, maître de conférence à l'Université de Rennes 1, pour la grande disponibilité dont elle a toujours fait preuve. Son encadrement, ses conseils et son soutien m'ont été précieux, et c'est avec un grand plaisir et un grand intérêt que j'ai travaillé à ses cotés tout au long de ces trois années.

Je remercie Thierry Priol, directeur de recherche à l'INRIA, pour m'avoir accueilli dans son équipe et pour la confiance qu'il m'a toujours témoigné.

J'adresse mes plus vifs remerciements à Xavier Rousset de Pina, professeur à l'Institut National Polytechnique de Grenoble et à Willy Zwaenopoel, professeur à l'université de Rice, pour avoir accepté d'être rapporteurs de cette thèse et qui m'ont prodigué de judicieux conseils pour la mise en forme définitive de ce document.

Je remercie également Franck Cappello, chargé de recherche au CRNS, d'avoir accepté de participer à ce jury.

Je remercie vivement les membres de l'équipe Paris pour l'accueil qu'ils m'ont réservé et plus particulièrement Geoffroy Vallée, Pascal Gallard et Gaël Utard, qui ont supporté mes sautes d'humeur et mes excentricités durant cette dernière année.

Enfin, je tiens tout particulièrement à remercier ma famille, et notamment mes parents qui m'ont soutenu tout au long de mes études et sans qui je n'aurais pu aujourd'hui présenter ce travail.

TABLE DES MATIÈRES

1 INTRODUCTION GÉNÉRALE

Depuis l'apparition de l'ordinateur comme outil de calcul numérique, les scientifiques et ingénieurs, grands consommateurs de puissance de calcul, n'ont cessé de demander toujours plus de performance de la part des ordinateurs. Très rapidement, l'utilisation d'une unique unité de calcul s'est révélée insuffisante pour répondre aux exigences de performance attendues. L'apparition des calculateurs parallèles à la fin des années 60, a permis de multiplier la puissance de calcul des ordinateurs en augmentant le nombre de processeurs. Ces premières versions de calculateurs parallèles étaient composées d'un ensemble d'unités de calcul (processeur/mémoire) reliées par un réseau d'interconnexion. Néanmoins, tout en offrant une solution aux besoins de haute performance, ces machines posaient de nouveaux problèmes, notamment d'ordre logiciel. Comment programmer ces architectures dont les multiples ressources sont distribuées sur les nœuds du calculateur ? Les méthodes utilisées jusqu'alors sur les architectures séquentielles de type Von Neuman ne s'appliquent plus à ces architectures.

Contexte historique

La programmation parallèle par échange de messages a été la première réponse au problème de la programmation de ces architectures. Cependant, en utilisant ce paradigme la tâche du programmeur devient beaucoup plus complexe, puisque celui-ci doit gérer le découpage de l'algorithme en sous-algorithmes pouvant s'exécuter en parallèle. De plus, il doit gérer manuellement le placement des tâches de calcul sur les différents nœuds et les échanges de données entre les différentes unités de calcul. Très rapidement, il est apparu que ce type de programmation demandait des compétences pointues, que seuls des spécialistes pouvaient réellement maîtriser.

L'inconvénient majeur de ces calculateurs parallèles est la distribution des ressources, notamment la ressource mémoire. L'apparition des architectures à mémoire partagée a permis de simplifier de manière significative la tâche du programmeur. Différentes solutions ont été proposées afin d'offrir une mémoire partagée. On peut citer les mémoires à accès non uniforme ou « *NUMA (Non Uniform Memory Access)* », permettant à un processeur de pouvoir accéder à sa mémoire locale à travers un bus et à la mémoire des nœuds distants à travers le réseau d'interconnexion. Cette différence d'accès aux mémoires induit une importante variation du temps d'accès aux données suivant leur localisation dans la machine, aboutissant à de nouvelles formes de contraintes de programmation.

Les architectures à mémoires attractives ou « *COMA (Cache Only Memory Architec-*

ture) », résolvent en partie ce problème. Celles-ci sont composées d'un ensemble de nœuds disposant d'une mémoire locale dans laquelle les données référencées par le processeurs sont dupliquées. Enfin, les architectures à accès uniforme aux données ou « *UMA (Uniform Memory Access)* » offrent une mémoire unique partagée par plusieurs processeurs, libérant ainsi le programmeur d'applications parallèles des contraintes liées à la gestion mémoire. L'extensibilité de ces architectures est cependant limitée par des contraintes matérielles.

L'apparition de ces architectures à mémoire partagée a permis de passer d'un modèle de programmation par échange de messages à un modèle de programmation par *threads* et variables partagées, beaucoup plus simple puisque libéré des contraintes liées à la gestion des mémoires distribuées et à la communication entre les tâches de calcul.

Parallèlement à cette évolution matérielle des calculateurs parallèles, les systèmes d'exploitation conçus pour ces machines ont évolué afin de mieux gérer les ressources disques et processeurs. Des mécanismes permettant l'ordonnancement de processus sur l'ensemble des processeurs d'un calculateur parallèle offrent une meilleure répartition de la charge de calcul tout en masquant la configuration matérielle de ces processeurs. De la même manière, les systèmes de fichiers parallèles permettent d'utiliser au mieux la ressource disque en offrant la vision d'un disque unique disposant d'un volume de stockage et d'une bande passante très élevée.

L'ensemble de ces progrès technologiques, tant d'un point de vue matériel que logiciel, permet de disposer aujourd'hui de calculateurs parallèles performants et simples d'utilisation. Subsistent cependant plusieurs inconvénients, dont le plus flagrant est certainement le coût financier, se chiffrant en dizaines ou centaines de millions de francs. Le coût exorbitant de ces calculateurs limite leur diffusion à quelques grandes entreprises et laboratoires de recherche fortunés. Le second problème lié à ces architectures est le manque d'évolutivité du matériel. Les calculateurs parallèles, sont généralement conçus autour d'un processeur spécifique, qu'il est difficile, voire impossible de faire évoluer. En conséquence, les calculateurs parallèles ne peuvent suivre l'évolution rapide de la technologie des processeurs. Enfin, les machines parallèles à mémoire partagée, offrant un modèle de programmation simple, sont peu extensibles. Pour des raisons de complexité matérielle et de performance, il est très difficile ou déraisonnable d'utiliser plus de quelques dizaines de processeurs. Ces problèmes sont intrinsèques à ces architectures haut de gamme, qui nécessitent une conception et une mise au point longue et complexe, remise en cause à chaque nouvelle génération.

En marge du développement de ces calculateurs haut de gamme, on assiste depuis quelques année à l'émergence d'une nouvelle forme d'architecture : les réseaux de stations de travail et grappes de calculateurs. Nous nous focalisons dans ce document sur les grappes de calculateurs. Ces architectures sont composées d'un ensemble d'ordinateurs standard, interconnectés par l'intermédiaire d'un réseau local à haute performance. L'apparition de ces architectures est le fruit de la rapide évolution des technologies de réseaux et processeurs dans le domaine de l'informatique grand public. La technologie des réseaux locaux a vu sa bande passante multipliée par 100 en moins de 10 ans (Ethernet 10 megabits début 1990, à Myrinet et gigabit Ethernet vers 1998) tout en divisant la latence d'un facteur 1000. La technologie des processeurs a connu une évolution encore plus fulgurante en passant de processeurs séquentiels cadencés à quelques dizaines de méga-hertz en 1990

à des processeurs super-scalaires à exécution dans le désordre, dont la fréquence dépasse maintenant le Giga-Hertz. Cette évolution rapide de la technologie informatique grand public permet de disposer aujourd'hui d'ordinateurs personnels de haute performance pour un coût modique. La multiplication de ces ordinateurs autour d'un réseau local haute performance offre une puissante de calcul potentielle comparable à celle des calculateurs parallèles et ceci pour un coût d'un ordre de grandeur inférieur. Le prix d'un calculateur parallèle *Silicon Graphics* Onyx 2 équipé de six processeurs est de l'ordre de 1 million de francs, à comparer aux 150 000 francs[1] d'une grappe équipée de six processeurs de puissance équivalente et d'un réseau Gigabit Ethernet. Ces architectures représentent donc désormais une alternative attrayante aux calculateurs parallèles. Leur coût raisonnable, permet à des industries qui ne pouvaient se permettre l'achat de calculateurs parallèles, de se doter d'une puissance de calcul élevée. Ce phénomène, permet entre autre l'expansion de la simulation numérique, outil désormais indispensable à la conception de produits de haute technologie.

Problématique

L'enthousiasme initial autour des grappes d'ordinateurs a rapidement fait place à une certaine méfiance lorsque les utilisateurs ont pris conscience du bond technologique en arrière que représente ces architectures. Des calculateurs parallèles à mémoire partagée dotés de systèmes d'exploitation spécialisés offrant une grande simplicité d'utilisation, on revenait à des architectures comparables aux calculateurs parallèles de la fin des années 70, aux ressources totalement distribuées et équipés d'un système d'exploitation standard. La conséquence de ce bond en arrière est le retour à des modèles de programmation par échange de messages et une gestion explicite par le programmeur des ressources distribuées.

Pour simplifier la programmation des grappes et se rapprocher des fonctionnalités offertes par les calculateurs parallèles modernes, de nombreuses approches sont proposées par la communauté scientifique. L'objectif de ces travaux est de fournir à l'utilisateur la vue d'une machine unique à haute performance au dessus d'une grappe : un *système à image unique*. Différentes approches se dégagent, généralement liées à la gestion d'une ressource particulière de la grappe. Les mécanismes de *Mémoire Partagée Réparties* (MPR) ou de pagination à distance offrent la vision d'une mémoire unique et partagée au dessus d'un ensemble de modules mémoires distribués. Les *Systèmes de Gestion de Fichiers Distribués* (SGFD) masquent la distribution des disques grâce à une interface unique d'accès aux fichiers locaux et distants, tandis que les *Systèmes de Gestion de Fichiers Parallèles* (SGFP) distribuent le stockage des données d'un fichier sur un ensemble de disques, offrant la vision d'un disque unique de grande capacité et doté d'une bande passante élevée. Enfin, les mécanismes de migration et d'ordonnancement global autorisent l'exécution de processus sur l'ensemble des processeurs d'une grappe, de manière transparente pour l'utilisateur, tout en assurant un partage équitable de la charge de calcul sur l'ensemble des processeurs.

Les solutions proposées assurent la vision unique d'une ressource distribuée, mais leur

[1] Prix estimé d'achat en 1999

conception repose sur des concepts complexes et propres à chacune de ces ressources. La mise en œuvre est de plus très délicate. Une conséquence de cette complexité est l'absence de système offrant une gestion globale et une vision unique de toutes les ressources d'une grappe. La conception et la mise en œuvre d'un tel système sur la base des solutions existantes aboutirait à un système très complexe pour lequel il serait difficile de garantir une bonne fiabilité et de hautes performances.

Objectif de l'étude

L'objectif de cette thèse est de permettre la conception et la mise en œuvre d'un système d'exploitation assurant une gestion globale de toutes les ressources d'une grappe (processeurs, mémoires et disques) afin d'offrir la vision d'un multiprocesseur à mémoire partagée. Ce système doit permettre d'exécuter efficacement des applications multithreadées conçues pour un calculateur de type SMP sans aucune modification. Le premier volet de cette étude porte sur la conception d'un mécanisme générique de niveau noyau offrant les fonctionnalités nécessaires à la mise en œuvre des services systèmes de gestion globale des ressources. Ce mécanisme nommé *conteneur*, est fondé sur une gestion globale de la mémoire physique des nœuds d'une grappe. Il regroupe les deux principaux mécanismes utilisés pour la conception des systèmes existants de gestion globale des ressources, que sont : la migration / duplication de pages et la gestion de la cohérence de données dupliquées.

Le second volet de l'étude porte sur la conception et la mise en œuvre de services systèmes distribués de haut niveau, fondés sur l'utilisation des conteneurs. Nous montrons de quelle manière il est possible de réaliser très simplement, grâce aux conteneurs, des services aussi complexes qu'une mémoire virtuelle partagée, un système de gestion de fichiers distribué, un cache de fichier distribué coopératif ou un système distribué de projection de fichiers.

Le mécanisme de conteneur que nous proposons a été expérimenté dans le cadre de la conception du système d'exploitation baptisé Gobelins. Dans ce système, les conteneurs représentent le cœur de la gestion globale des ressources et permet d'offrir la vision d'un multiprocesseur à mémoire partagée au dessus d'une grappe de PCs.

Plan du document

Ce document est divisé en trois parties. La première partie est consacrée à la présentation des grappes d'ordinateurs et aux mécanismes de gestion globale des ressources. Le premier chapitre offre une introduction aux grappes de calculateurs et présente les technologies couramment utilisées sur ces architectures. Les chapitres suivants dressent un état de l'art des différentes technologies existantes pour une gestion globale des ressources distribuées. Nous focalisons notre discours sur la gestion des trois principales ressources d'un ordinateur que sont le processeur, la mémoire et les disques.

La seconde partie présente de quelle manière il est possible de factoriser les mécanismes utilisés par les différents systèmes de gestion globale des ressources, au sein d'un mécanisme

unique : les conteneurs. Le chapitre 7 présente la démarche que nous avons suivie dans cette thèse. Nous présentons dans le chapitre 8, la conception des conteneurs, et détaillons dans le chapitre 9 leur utilisation pour la mise en œuvre de services systèmes distribués de haut niveau.

Enfin, nous présentons dans la troisième partie les résultats obtenus lors de la mise en œuvre des conteneurs dans le système d'exploitation GOBELINS. Le chapitre 10 décrit la mise en œuvre du système d'exploitation GOBELINS, en mettant en évidence la légèreté des modifications à apporter à un système d'exploitation existant pour proposer un système à image unique reposant sur le concept de conteneurs. Nous présentons enfin au chapitre 11, les résultats d'une étude détaillée des performances de GOBELINS sur une grappe de PCs. Nous avons étudié plus particulièrement les performances d'applications parallèles utilisant la mémoire partagée du système GOBELINS, ainsi que les performances de son cache de fichiers coopératif.

Le chapitre 12 dresse un bilan de notre étude et ouvre les perspectives de notre travail.

Première partie

Grappes de calculateurs et gestion globale des ressources

2 GRAPPES DE CALCULATEURS

L'explosion du marché de l'Internet et l'utilisation de plus en plus fréquente de la simulation numérique dans l'industrie nécessite l'utilisation de calculateurs puissants. Les serveurs ou caches web nécessitent des espaces de stockage très importants, tandis que les serveurs de données Internet et les algorithmes de calcul numérique nécessitent une grande puissance de calcul. Les architectures de type super-calculateurs ont été conçues pour répondre à ce besoin de haute performance. Néanmoins, le coût financier de ces machines est tel qu'il ne peut être assumé par une petite ou moyenne entreprise. Cependant, les PME/PMI représentent un nouveau marché pour la haute performance. Pour cette cible particulière, les grappes de calculateurs constituent une alternative intéressante aux super-calculateurs. Une grappe est d'un coût financier très inférieur à celui d'un super-calculateur pour une performance potentielle quasiment identique. Suivant les architectures et les classes d'applications considérées, une grappe se révèle de 3 à 10 fois moins chère qu'un super-calculateur pour des performances identiques. Néanmoins, l'architecture totalement distribuée des grappes et le manque de logiciels adaptés en font des architectures difficiles à utiliser et à programmer. Pour simplifier la programmation des grappes, il est possible d'utiliser des solutions logicielles afin de cacher la multiplicité et la distribution des ressources au programmeur, lui donnant l'illusion de travailler sur une machine centralisée à très haute performance. La classe de logiciels offrant cette illusion de machine unique au dessus d'une grappe est connue sous le nom de **système à image unique**.

Nous présentons dans la suite de ce chapitre une définition détaillée de la notion de grappe de calculateurs et nous situons cette architecture par rapport aux différentes architectures à haute performance existantes. Nous détaillons ensuite les problèmes soulevés par la distribution des ressources au sein d'une grappe et donnons une définition d'un système à image unique.

2.1 Définitions

Depuis une vingtaine d'années, de nombreuses formes d'architectures à haute performances ont vu le jour. On peut cependant distinguer trois grandes familles d'architectures [17] :

– Les **architectures massivement parallèles** (MPP[1]), se caractérisent par un grand nombre de processeurs, une mémoire distribuée et un réseau d'interconnexion à très

[1]Massively Parallel Processors

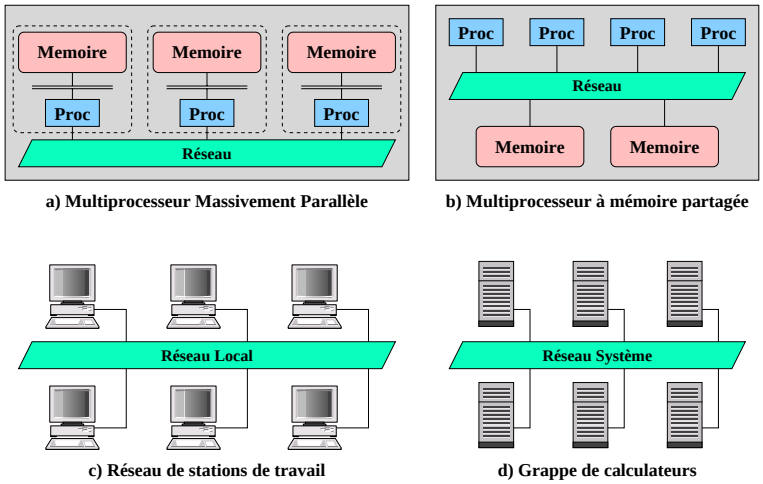

FIG. 2.1 – *Comparaison des différentes architectures parallèles et distribuées*

haute performance (c.f. figure 2.1.a).

- Les **architectures à mémoire partagée** de type multiprocesseur symétrique (SMP[2]) ou à accès non uniforme aux données (CC-NUMA [3]). Ces architectures sont caractérisées par un nombre limité de processeurs et des ressources totalement partagées par les processeurs (c.f. figure 2.1.b).
- Les **architectures distribuées** caractérisées par un nombre potentiellement grand de processeurs, des ressources totalement distribuées et un réseau d'interconnexion dont les performances peuvent varier de manière significative, suivant la technologie utilisée.

Au sein des architectures distribuées, on peut distinguer deux sous-catégories : les réseaux de stations de travail ou *Network Of Workstation (NOW)* [6] (c.f. figure 2.1.c) et les grappes de calculateurs ou *Cluster Of Workstations (COW)* (c.f. figure 2.1.d). Le tableau 2.1 présente un comparatif rapide de ces architectures.

	Nombre de proc.	Type de réseau	Débit du réseau
MPP	O(100)-O(1000)	Propriétaire	1 - 3 Gb/s
SMP/CC-NUMA	O(10)-O(100)	Mémoire partagée	3 - 10 Gb/s
NOW	O(10)-O(1000)	LAN	10 - 100 Mb/s
COW	O(10)-O(1000)	SAN	100 Mb/s - 2 Gb/s

TAB. 2.1 – Comparaison de différentes architectures multiprocesseurs hautes performances

[2]Symetric Multi Processor
[3]Cache Coherent - Non Uniform Memory Access

2.1.1 Réseau de stations de travail

Définition 1 (LAN) *Un **LAN** (Local Area Network) est un réseau reliant des machines géographiquement réparties au sein d'une entreprise ou d'une institution. Les distances entre les machines s'expriment en dizaines ou centaines de mètres.*

Définition 2 (NOW) *On désigne par **NOW** un ensemble de stations de travail géographiquement distribuées au sein d'une entreprise ou d'une institution et interconnectées par un réseau local (LAN). Chaque station est utilisée en mode interactif par un utilisateur privilégié.*

Dans un NOW, les stations de travail sont généralement sous-utilisées. En mode interactif, l'utilisateur réalise généralement de l'édition de texte et utilise peu le processeur et la mémoire. Lorsqu'il s'absente, la station est totalement inutilisée. L'objectif des architectures NOW et des systèmes associés, tel GLUnix [43], est d'éviter le gaspillage de ressources en permettant une utilisation distribuée et transparente des ressources inutilisées. Les ressources processeurs, mémoires et disques disponibles à travers le réseau sont mises en commun et peuvent être allouées pour un utilisateur ayant temporairement un besoin important d'une ou plusieurs de ces ressources.

Ces architectures sont soumises à deux contraintes majeures. La première concerne les performances du réseau local, généralement de type Ethernet 100 megabits. Les performances modestes de cette technologie conditionnent les algorithmes en terme de volume de données échangées et de temps de réponse des services distribués mis en jeu. La deuxième contrainte, et certainement la plus importante en terme de popularité des architectures NOW [6], est de ne pas dégrader les performances du système pour l'utilisateur travaillant en mode interactif. Cet utilisateur est prioritaire pour l'utilisation des ressources de sa machine. Par exemple, si celles-ci ont été allouées à un autre utilisateur en son absence, elles doivent lui être restituées intégralement à son retour.

2.1.2 Grappe de calculateurs

Définition 3 (SAN) *Un **SAN** (System Area Network) est un réseau reliant les nœuds d'une grappe. Les distances entre les machines s'expriment en mètres.*

Définition 4 (Grappe) *On désigne par **COW** ou **grappe de calculateurs** un ensemble de calculateurs (stations de travail, PCs) interconnectés par un réseau système (SAN). Les calculateurs ne sont pas utilisés en mode interactif et sont géographiquement regroupés au sein d'un local spécialisé.*

Définition 5 (nœud) *On appelle **nœud** un calculateur d'une grappe.*

L'objectif des grappes de calculateurs est de proposer une alternative aux calculateurs parallèles pour le calcul haute performance. Contrairement aux architectures NOWs, on ne cherche pas à puiser la puissance de calcul au sein des ressources inutilisées d'un parc de stations de travail déjà installé, mais à construire une machine de haute performance à

base d'un ensemble de calculateurs dédiés et bon marché.

Au sein d'une grappe, il est possible d'utiliser des technologies réseaux plus performantes que dans les NOWs : les réseaux systèmes. Ce type de réseau est souvent sujet à des contraintes matérielles rendant difficile, voire impossible la dispersion des machines à travers un bâtiment. De plus, équiper un parc de stations de travail d'un réseau système à très haut débit n'est généralement pas justifié. Au sein d'une grappe, où les calculateurs sont réunis dans un local spécialisé, il est possible et justifié d'utiliser des réseaux systèmes dans un but de haute performance.

Enfin, en supprimant la notion d'utilisateur privilégié, la gestion des ressources distribuées est simplifiée. L'existence d'un utilisateur privilégié est en effet une contrainte forte dans les architectures NOW. Le moment où un utilisateur privilégié va demander l'utilisation de ressources et la nature de ces ressources est totalement imprévisible. En supprimant cette contrainte dans les grappes il est possible de mettre en œuvre des mécanismes de gestion des ressources distribuées plus efficaces et plus évolués. Un aperçu de ces mécanismes est présenté dans les chapitres suivants.

2.2 Communication au sein d'une grappe

La communication au sein d'une grappe est un élément essentiel puisque les performances de l'architecture sont intimement liées aux performances de la technologie réseau et des bibliothèques de communications utilisées. Nous présentons dans cette partie, un aperçu des technologies matérielles et des bibliothèques de communication couramment utilisées sur les grappes de calculateurs.

2.2.1 Réseaux d'interconnexion

De nombreuses technologies réseaux permettent de construire des réseaux locaux ou des réseaux systèmes [17]. Nous présentons ici, les trois technologies les plus fréquemment utilisées dans le domaine des grappes de calculateurs.

2.2.1.1 Ethernet

Ethernet est sans aucun doute la technologie réseau la plus utilisée, avant tout pour des raisons historiques. Ethernet a été la première technologie d'interconnexion grand public, offrant des débits de 1Mb/s dès les années 70. De nombreuses entreprises se sont équipées de cartes offrant un débit de 1 ou 10 Mb pour construire leur réseau local d'entreprise. Lors de l'apparition de technologies offrant des débits plus élevés (100 Mb puis 1 Gb) la plupart des utilisateurs de la technologie Ethernet ont préféré continuer à utiliser celle-ci plutôt que les nouvelles technologies concurrentes (notamment ATM et Myrinet), principalement pour des raisons de compatibilité et de coût.

Un réseau Ethernet est typiquement composé d'un ensemble de cartes d'interface réseau reliées par des répéteurs (*hub*). Les répéteurs se comportent comme des bus en relayant tout

message arrivant vers toutes les cartes connectées. Les répéteurs peuvent être reliés entre eux suivant un grand nombre de topologies. Les câbles assurant les liaisons carte/répéteur ou répéteur/répéteur peuvent atteindre une longueur de l'ordre de quelques kilomètres. Une plus grande distance peut être couverte en utilisant des répéteurs.

2.2.1.2 Myrinet

Myrinet [14] est une technologie récente, issue du monde des calculateurs massivement parallèles. Un réseau Myrinet est composé de liens de communication point à point reliant entre eux les commutateurs et les nœuds finaux du réseau. Les câbles assurant la liaison entre les cartes et les commutateurs peuvent avoir une longueur comprise entre 1 à 200 mètres. Cette technologie offre des débits de 2 Gbit/s sur chaque lien, tout en garantissant des latences de l'ordre de quelques micro-secondes. Myrinet est avant tout une technologie conçue pour construire des réseaux systèmes, mais il est possible d'utiliser Myrinet dans le cadre de petits réseaux locaux.

Les cartes d'interface réseau sont équipées d'un processeur et d'une mémoire SRAM. Cette mémoire (de 2 à 8 Mo) sert à la fois de tampon pour la gestion des communications et de support pour les variables et le code du programme de contrôle : le MCP [4]. Le MCP est chargé par l'hôte dans la mémoire de la carte réseau et s'exécute dès que le pilote de la carte en donne l'ordre. L'originalité de cette technologie est la possibilité de modifier ce programme de gestion bas niveau des échanges réseaux pour l'adapter à des besoins particuliers [13].

2.2.1.3 SCI

Les réseaux à capacité d'adressage (« *RMA : Remote Memory Access* ») offrent un nouveau modèle de communication en permettant d'accéder directement aux données situées dans la mémoire d'une machine distante grâce aux opérations de lecture et écriture du processeur, à la manière des architectures NUMA. Les trois principales mises en œuvre de ce type de réseau sont SCI [46], « *Memory Channel* » [44] et VMMC [34]. En pratique, ces réseaux permettent de projeter une partie de la mémoire locale d'une machine dans l'espace d'adressage d'entrées/sorties d'une autre machine.

La technologie « *Scalable Coherent Interface (SCI)* »[46] est actuellement la mise en œuvre la plus aboutie d'un réseau à capacité d'adressage. Un réseau SCI est composé de liens de communication reliant les cartes autour d'un anneau. Ces anneaux peuvent être connectés grâce à des commutateurs. La longueur des liens de communication est limitée à quelques mètres. Cette technologie offre des débits de 1,6 Gbit/s sur chaque lien, tout en garantissant des latences de l'ordre de quelques micro-secondes. La technologie SCI est presque exclusivement réservée à la construction de réseaux systèmes.

L'originalité du réseau SCI est de permettre à une machine d'accéder directement à la mémoire d'une autre machine, ceci grâce à un simple accès mémoire, sans demander explicitement de communication. Une connexion SCI est réalisée par le pilote en projetant

[4] Myrinet Control Program

une plage d'adresses physiques correspondant à un canal d'entrée/sortie géré par la carte, dans l'espace d'adressage virtuel d'un processus. Les accès aux adresses correspondant à une zone d'adressage SCI sont interceptés par la carte et redirigés vers une carte distante qui effectue l'accès à l'adresse correspondante en mémoire physique.

Cette technologie offre deux avantages majeurs. Le premier est la grande simplicité de programmation des applications distribuées, car la gestion des communications n'est plus à la charge du programmeur, mais réalisée automatiquement et de manière transparente par le réseau. Le second avantage est de pouvoir se passer de bibliothèque de communication et ainsi de supprimer tous les surcoûts logiciels intervenant lors d'une communication traditionnelle. Les performances, notamment en terme de latence, s'en trouvent grandement améliorées. Cependant, la différence de latence lors de l'accès à des données locales ou distantes[5] a une grande infuence sur les performances et ne doit pas être négligée lors de la mise en œuvre d'applications.

2.2.2 Bibliothèques de communication

Actuellement, la programmation des grappes de calculateurs repose presque exclusivement sur le paradigme d'échange de messages. Cette partie présente les bibliothèques de communication les plus couramment utilisées.

2.2.2.1 TCP/IP et sockets

Internet Protocol (IP) est sans aucun doute le protocole réseau le plus répandu et le plus utilisé. Ce protocole fourni un transfert point à point, sans connexion et non fiable. La couche IP n'a pas été conçue pour être accédée directement par les utilisateurs, mais pour servir de base à des protocoles de plus haut niveau, dont les deux principaux sont *Transmission Control Protocol (TCP)* et *User Datagram Protocol (UDP)*. TCP/IP est certainement le protocole le plus utilisé. Il offre des transmissions fiables en mode connecté. Ces protocoles sont accessibles à travers une interface standard connue sous le nom de *sockets*. L'interface *socket* a permis une intégration élégante de la couche réseau dans le monde Unix™, en proposant un mécanisme d'accès au réseau fondé sur la traditionnelle abstraction fichier.

Cependant, les *sockets* souffrent de nombreux points faibles dans le cadre d'une utilisation pour le calcul haute performance au sein d'une grappe. Tout d'abord, l'interface offerte est lourde et peu adaptée au modèle de programmation utilisé pour la conception d'applications parallèles. Toute communication avec une machine nécessite l'ouverture préalable d'une connexion, multipliant ainsi le nombre de connexions à gérer. De plus, aucun mécanisme de désignation simple des nœuds ou d'identification des messages n'est offert. Enfin, l'architecture en couches du protocole TCP/IP, implique de nombreuses recopies et formatages des données lors de l'émission et de la réception, ce qui limite l'utilisation optimale de la bande passante du réseau et induit des latences élevées.

[5] Quelques nanosecondes pour accéder au cache local, quelques microsecondes pour accéder à une donnée distante.

Afin d'offrir de meilleures performances et des interfaces plus adaptées à la programmation d'applications parallèles, de nombreuses bibliothèques spécialisées ont vu le jour. La suite de ce paragraphe en offre un aperçu.

2.2.2.2 PVM - MPI

Parallel Virtual Machine (PVM) [97] a été la première bibliothèque de communication utilisable dans le contexte des architectures distribuées. Elle fournit une interface de programmation simple pour la création de tâches de calcul et la communication entre ces tâches et permet l'utilisation de machines et de systèmes hétérogènes au sein d'un même espace de communication. Les processus communiquants sont identifiés grâce à des numéros logiques. Un type peut être affecté à chaque message permettant de lui associer une sémantique particulière. Enfin, PVM offre des fonctions de synchronisation de processus distribués (barrières, etc).

Néanmoins, si PVM offre une interface simple et une excellente portabilité, la plupart des mises en œuvre n'offrent pas les performances attendues par les programmeurs d'applications parallèles. Il existe cependant des mises en œuvres efficaces, notamment sur les calculateurs parallèles, offrant des performances élevées.

Dans le domaine des applications parallèles, *Message Passing Interface (MPI)* [40] est une bibliothèque de communication plus récente, qui est désormais le standard *de facto* pour la communication dans les architectures distribuées. MPI offre une interface beaucoup plus riche que PVM, constituée d'une profusion de fonctions de communication, synchronisation, etc. MPI permet également l'accès à des unitées de stockage distantes, grâce à l'extension MPI-IO[25]. Les fichiers distants sont accédés à travers un modèle d'envoi-réception de messages : une réception correspond à une lecture alors qu'un envoi correspond à une écriture.

2.2.2.3 Messages Actifs

Eicken *et al* [96] proposent une alternative à l'interface classique de type envoi-réception de messages. Les messages actifs fonctionnent sur un modèle de communication unilatérale. Dans ce modèle de communication, l'émission d'un message peut avoir lieu à n'importe quel moment, sans tenir compte de l'activité du nœuds destinataire. La primitive de réception est remplacée par un mécanisme d'activation de fonction. A chaque type de message est associée une fonction exécutée automatiquement lors de l'arrivée d'un nouveau message.

L'objectif des messages actifs est de réduire les surcoûts de communication, liés à la gestion des messages reçus. Avec une interface de type envoi-réception, les messages reçus sont généralement recopiés dans un tampon avant d'être consommés lors d'un appel à une fonction de réception. Dans le cas des messages actifs, un message est consommé dès son arrivée, ce qui permet de supprimer les recopies en réception et de diminuer le temps nécessaire à la prise en compte d'un message reçu. Il en découle une augmentation de la bande passante et une diminution significative de la latence.

2.2.2.4 BIP - Gamma

BIP [82] est une bibliothèque de communication haute performance dédiée à la technologie Myrinet. Elle fournit une couche réseau de bas niveau, fondée sur des primitives d'envoi et de réception bloquantes aussi bien que non bloquantes. La conception de BIP permet d'offrir aux applications une interface d'accès direct au réseau, court-circuitant le noyau et permettant ainsi de diminuer la latence en supprimant les changements de contexte et la traversée de couches logicielles. De plus, une mise en œuvre de type zéro copie mémoire permet d'atteindre des débits proches des débits maximum théoriques du réseau Myrinet.

Bien que BIP ait été conçu avant tout pour servir de base à la mise en œuvre de bibliothèques de communication de plus haut niveau comme MPI [40], il est possible de l'utiliser directement au sein d'une application parallèle. Il est toutefois à noter qu'aucun mécanisme de contrôle d'erreur n'est mis en œuvre et que la carte réseau ne peut être utilisée que par un seul utilisateur à la fois.

Gamma [23] est une bibliothèque de communication haute performance dédiée à la technologie Ethernet. Elle reprend la philosophie de BIP (mise en œuvre bas niveau, zéro copie, accès direct à la carte depuis le niveau utilisateur), mais offre une interface de type messages actifs. Contrairement à BIP, Gamma a été conçue pour être utilisée par les programmeurs d'applications parallèles et dispose donc d'une interface plus étoffée.

Le tableau 2.2 présente un comparatif des performances obtenues avec différentes technologies d'interconnexion et différentes bibliothèques de communication.

Réseau	Interface	Plate-forme	Latence (μs)	Débit (Mo/s)
Ethernet 100	Linux TCP/IP sockets	Pentium II 350 MHz	58	11,1
Ethernet 100	MPI (LAM MPI)	Pentium II 500 MHz	129	11
Ethernet 100	Gamma	AMD K7 500 MHz	14,3	12,1
Ethernet 1000	Linux TCP/IP sockets	Pentium II 400 MHz	59	31
Ethernet 1000	MPI (LAM MPI)	Pentium II 450 MHz	140	37,5
Ethernet 1000	Gamma	Pentium II 450 MHz	9,5	93,7
Myrinet	Fast Sockets	Ultra Sparc	57,8	32,9
Myrinet	MPI (MPICH)	Pentium II 500 MHz	15	94,4
Myrinet	BIP	Pentium Pro	4,3	126
Myrinet 2000	BIP	Pentium III 1 GHz	1,7	245
SCI	Accès direct	Pentium II 450 MHz	2	90
SCI	TCP/IP sockets	Pentium II 450 MHz	8	72
SCI	MPI (MPICH)	Pentium II 450 MHz	7	75

TAB. 2.2 – Comparaison des performances crêtes mesurées sur différents réseaux et bibliothèques de communication

2.2.2.5 RPC - Corba

Les mécanismes d'activation de procédures à distance tel les RPC[6] de SUN, offrent une certaine abstraction du réseau. L'interface d'envoi/réception laisse ici la place à des mécanismes plus complexes permettant d'activer un service distant par un simple appel de procédure. Ce mécanisme nécessite un enregistrement préalable des services pouvant être exécutés à distance, des procédures réalisant l'activation des services et du format des données échangées. Lors d'un appel à l'une de ces procédures, le mécanisme de RPC réalise un empaquetage des données passées en argument, et les transmet à la machine hébergeant le service désiré. Un gestionnaire de RPC reçoit les données, les extrait et active le service. Celui-ci peut éventuellement retourner un résultat en suivant un schéma similaire.

Le RPC est devenu un standard *de facto* pour la mise en œuvre d'applications distribuées de type client-serveur. Il souffre cependant d'un problème de performance, lié aux surcoûts induits par l'empaquetage/dépaquetage des données et à l'activation de gestionnaires.

Corba [75] est une norme récente offrant des services comparables aux RPC, mais dans un contexte de programmation par objets. L'objectif de Corba est de permettre la conception d'applications distribuées fondées sur des mécanismes d'activation de méthodes à distance. Même si un effort très important a été consenti afin de fournir des transferts de données rapides et des latences raisonnables, la plupart des mises en œuvre offrent des performances médiocres. Des travaux récents [32] proposent néanmoins un CORBA à haute performance, offrant des latences et débits comparables à ce qui existe dans les bibliothèques de communication à haute performance.

2.3 Systèmes à image unique

La grande variété de bibliothèques de communication sur grappes est symptomatique du besoin de concilier performance et simplicité d'utilisation. De plus, le modèle de programmation dominant le développement d'applications sur les architectures distribuées est sans conteste l'échange de messages. Comme nous l'avons déjà évoqué, ce modèle de programmation, qui fut le premier à être utilisé sur les machines parallèles, introduit de nouvelles difficultés lors de la conception d'applications parallèles ou distribuées. Pour remédier à ce problème, les concepteurs de calculateurs parallèles se sont tournés vers des solutions à base de mémoire partagée offrant un modèle de programmation plus simple et plus proche de la programmation de machines séquentielles. Depuis une vingtaine d'années, on peut observer le même phénomène dans le domaine des architectures distribuées et plus récemment dans le domaine des grappes. De nombreux travaux sont effectués dans la recherche et l'industrie afin de simplifier la programmation des grappes. L'objectif de ces travaux est d'offrir la vision d'une machine unique au dessus d'un ensemble de machines distribuées : un **système à image unique** ou **Single System Image (SSI)**.

[6] Remote Procedure Call

2.3.1 Définition

Dans ce document, nous distinguons deux types de ressources : les ressources physiques (processeur, disque dur, mémoire, etc) et les ressources logiques (processus, fichier, données en mémoire, etc) (voir figure 2.2).

Définition 6 (Ressource physique) *Une **ressource physique** est un élément matériel constituant un ordinateur qui peut être utilisé par un élément logiciel. Une ressource physique est indissociable du nœud sur lequel elle se trouve et ne peut être déplacée.*

Définition 7 (Ressource logique) *Une **ressource logique** est un élément logiciel qui peut être utilisé par un autre élément logiciel. Une ressource logique nécessite la présence d'une ou plusieurs ressources physiques pour exister. Une ressource logique peut être utilisée à distance ou déplacée d'un nœud vers un autre.*

Définition 8 (Système à image unique) *Un **système à image unique** est un dispositif matériel ou logiciel créant l'illusion qu'un ensemble de ressources (logiques ou physiques) distribuées ne forment qu'une ressource unique et partagée.*

L'objectif d'un SSI est double. Il doit rendre transparent l'accès aux ressources distantes tout en permettant d'exploiter au mieux l'ensemble des ressources d'une grappe. Un SSI peut être réalisé à différents niveaux : matériel, système d'exploitation, intergiciel[7] ou application.

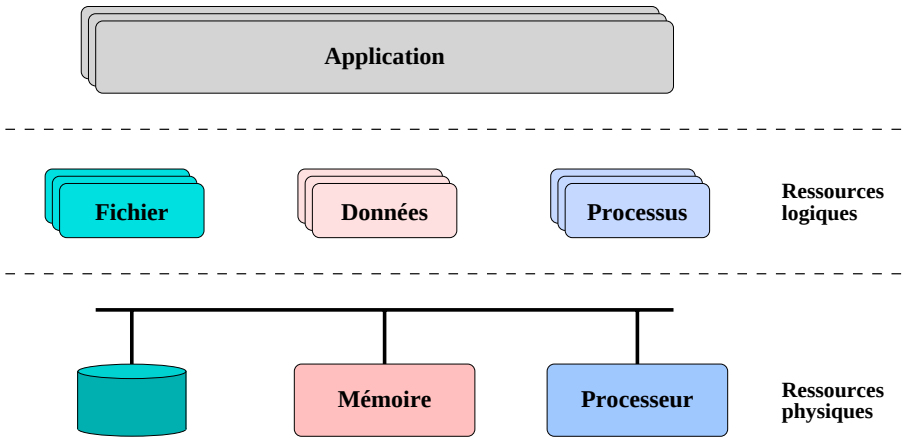

FIG. 2.2 – *Ressources physiques et logiques au sein d'un calculateur*

Les mises en œuvre matérielles reposent sur du matériel spécifique permettant de partager une ressource à travers un ensemble de nœuds. On peut citer SCI [46] ou Digital Memory Channel [44] qui offrent une mémoire partagée de type NUMA au sein d'une grappe, ou encore SCSI [1] ou Fiber Channel [5] qui offrent un accès partagé à un ensemble de disques.

[7] middleware

Les mises en œuvre de niveau noyau de système d'exploitation consistent principalement en des extensions de systèmes existants (Solaris, Linux, Windows NT) permettant de partager tout ou partie des ressources d'une grappe. On peut citer GLunix[43], Solaris MC[57], Sprite[76], Mosix[3], etc.

Enfin, les mises en œuvre de niveau utilisateur, qu'elles soient de type intergiciel, c'est-à-dire situées entre le système d'exploitation et les applications, ou de niveau applicatif via l'utilisation de bibliothèques spécialisées, représentent la classe la plus simple à mettre en œuvre, mais également la classe la plus restrictive du point de vue des fonctionnalités offertes et de la souplesse d'utilisation, puisque très loin du matériel. On peut citer Condor[4], Ivy[64], ThreadMarks[55], Millipede[41], etc.

Nous nous intéressons principalement dans ce document aux mises en œuvre logicielles et plus particulièrement aux mises en œuvre de niveau système.

Un système à image unique idéal doit permettre de regrouper chaque type de ressource en une ressource unique et partagée. L'utilisateur ne doit pas avoir connaissance de la distribution et de la multiplicité des ressources, mais avoir la vision d'une machine unique à haute performance. Pour cela, un SSI doit répondre à quatre objectifs principaux :

- Transparence à la distribution ;
- Transparence à l'utilisation ;
- Sûreté de fonctionnement ;
- Extensibilité.

2.3.2 Transparence à la distribution

L'objectif fondamental d'un SSI est la disparition de la notion de nœud. Pour qu'un ensemble de machines apparaissent comme une machine unique, il ne doit pas être utile de nommer un nœud pour utiliser ses ressources. L'existence de multiples nœuds et la distribution des ressources doivent être totalement invisibles à l'utilisateur. Pour assurer une transparence totale du nombre et de la répartition des ressources, un SSI doit disposer des quatre propriétés suivantes :

Propriété 1 (Transparence de localisation) *Un SSI est* ***transparent à la localisation****, si la méthode d'accès à une ressource logique est la même que celle-ci soit mise en œuvre localement ou à distance.*

Propriété 2 (Indépendance de localisation) *Un SSI est* ***indépendant à la localisation****, s'il est possible de déplacer une ressource logique sans modifier la méthode d'accès à celle-ci.*

Propriété 3 (Partage logique) *Il y a* ***partage logique*** *au sein d'un SSI lorsque les ressources logiques d'une grappe peuvent être partagées par des applications s'exécutant sur des nœuds différents ou non.*

Propriété 4 (Partage physique) *Il y a* ***partage physique*** *au sein d'un SSI lorsque les ressources physiques d'une grappe peuvent être partagées par des applications s'exécutant sur des nœuds différents ou non.*

La propriété de **partage physique** n'est pas nécessaire pour offrir l'illusion d'une machine unique, mais elle permet de mieux utiliser l'ensemble des ressources d'une grappe. Si un processus a un besoin très important en terme de consommation mémoire ou disque, il peut être intéressant de lui permettre d'utiliser les ressources de l'ensemble de la grappe, plutôt que de le limiter aux ressources présentes sur son nœud d'exécution.

Pour qu'un système soit qualifié de *système à image unique*, il lui suffit de remplir les propriétés de transparence. Cependant, il est intéressant, voir indispensable d'introduire de la qualité de service au sein de ce système, notamment la transparence à l'utilisation, la sûreté de fonctionnement et l'extensibilité.

2.3.3 Transparence à l'utilisation

Pour exécuter une application sur un système à image unique il peut être nécessaire d'y intégrer du code spécifique au SSI, de la lier avec des bibliothèques dédiées, de la recompiler ou encore de la réécrire totalement pour ce système. Cette adaptation des applications représente un coût qui n'est pas toujours souhaitable. Afin de limiter ce coût, un SSI doit offrir la propriété de **transparence à l'utilisation**.

Propriété 5 (Transparence à l'utilisation) *Un SSI est **transparent à l'utilisation**, s'il ne nécessite pas l'utilisation explicite par l'utilisateur de mécanismes, bibliothèques ou éléments logiciels spécifiques à ce système.*

La propriété de transparence à l'utilisation n'est pas nécessaire pour fournir la vision d'une machine unique, mais elle permet une utilisation plus simple du système en évitant de modifier les habitudes de travail des utilisateurs et en limitant le surcoût de formation et d'apprentissage lié à l'utilisation d'un nouveau système.

Pour remplir la propriété de transparence à l'utilisation, un SSI mis en œuvre au niveau noyau doit offrir une interface et un environnement de programmation identique à celui d'un système existant, tel que Unix™ou Windows™par exemple. Un SSI mis en œuvre au niveau utilisateur ne doit pas modifier l'interface ou la sémantique des outils de conception d'applications (éditeurs de textes, compilateurs, lieurs, etc), ni les méthodes de programmation utilisées par les concepteurs d'applications.

2.3.4 Sûreté de fonctionnement

La multiplication des calculateurs et des ressources au sein d'une grappe augmente de manière significative le risque de défaillance de l'un des composants matériels. Si l'on considère une grappe comme une machine unique, la défaillance de l'un des composants peut signifier la défaillance de toute la grappe, ce qui n'est bien entendu pas souhaitable.

D'autre part, les applications de grande envergure, tant du point de vue de la taille des données manipulées que du temps d'exécution sont particulièrement sensibles à la défaillance d'un composant. La défaillance d'un composant signifie pour ces applications, la fin prématurée de leur exécution. La reprise de leur exécution depuis l'état initial représente

un gaspillage de ressource d'autant plus pénalisant que le temps d'exécution déjà écoulé est important.

Pour éviter ces deux situations, un SSI doit disposer de deux propriétés supplémentaires :

Propriété 6 (Haute disponibilité) *Un SSI est dit à* ***haute disponibilité*** *si la défaillance d'une ou de plusieurs ressources physiques n'empêche pas l'utilisation des ressources physiques restantes.*

Propriété 7 (Reprise d'applications) *Un SSI est dit à* ***reprise d'applications*** *si la perte d'une ou de plusieurs ressources physiques n'entraîne pas la perte définitive de ressources logiques.*

2.3.5 Extensibilité

Durant la vie d'une grappe, des nœuds peuvent disparaître définitivement suite à une défaillance permanente. D'autre part, les utilisateurs de la grappe peuvent désirer ajouter de nouveaux nœuds afin d'augmenter les ressources globales de la grappe pour répondre à de nouveaux besoins. Un SSI doit être en mesure de gérer la fluctuation du nombre de nœuds et de tirer le meilleur parti des ressources dont il dispose à un instant donné.

Propriété 8 (Extensibilité des performances) *Il y a* ***extensibilité des performances*** *dans un SSI, si l'augmentation du nombre de ressources physiques augmente proportionnellement les performances globales du système.*

Propriété 9 (Transparence à l'extensibilité) *Un SSI est* ***transparent à l'extensibilité*** *si lors de l'ajout et du retrait d'un nœud dans la grappe, ses ressources physiques sont automatiquement intégrées ou retirées de l'ensemble des ressources physiques disponibles. Ceci sans arrêter la grappe, de manière transparente pour l'utilisateur et avec un minimum de perturbation pour les applications en cours d'exécution.*

2.4 Résumé

Nous avons présenté dans ce chapitre une classification des architectures parallèles et distribuées. Les architectures parallèles se caractérisent par un grand nombre de processeurs reliés par un réseau d'interconnexion spécialisé à très haute performance. Les calculateurs parallèles récents possèdent des dispositifs de partage physique des modules mémoire répartis. Ces architectures très performantes sont également extrêmement coûteuses.

Les architectures distribuées sont composées d'un ensemble de machines interconnectées par un réseau local ou un réseau système. On désigne par grappe une architecture distribuée où les nœuds ne sont pas utilisés de manière interactive, mais uniquement comme ressource de calcul ou de stockage. Ces architectures sont généralement dix fois moins chères qu'un calculateur parallèle de puissance comparable. Cependant, les performances modestes du réseau d'interconnexion[8] et la distribution des ressources rend difficile l'utilisation optimale

[8] comparées aux réseaux internes des calculateurs parallèles

de la puissance potentielle d'une grappe.

L'augmentation rapide et continue de la performance des réseaux d'interconnexion et l'émergence de bibliothèques de communication à haute performance réduit de manière significative le fossé qui existe entre les performances des réseaux des architectures distribuées et parallèles. Néanmoins, les grappes apparaissent comme des architectures totalement distribuées dont la programmation est complexe.

Une solution pour simplifier la programmation des grappes est de masquer la distribution des ressources grâce à un système à image unique (SSI). Un SSI est un mécanisme matériel ou logiciel permettant de donner l'illusion qu'un ensemble de ressources distribuées ne forme qu'une seule ressource partagée. Un SSI idéal doit fournir quatre propriétés : (1) la transparence à la distribution, qui permet de masquer la distribution des ressources, (2) la transparence à l'utilisation, qui permet une utilisation simple du SSI, (3) la sûreté de fonctionnement permettant au SSI de continuer à fonctionner en présence de défaillances et (4) l'extensibilité permettant d'utiliser automatiquement les ressources ajoutées à une grappe.

Dans la suite de cette partie, nous étudions les systèmes à image unique sous l'angle de la propriété de transparence à la distribution et de transparence à l'utilisation. Les mécanismes de sûreté de fonctionnement et d'extensibilité ne sont pas abordés.

3 GESTION GLOBALE DE LA RESSOURCE MÉMOIRE

Si l'on considère un SSI fondé sur une gestion globale de la mémoire, les propriétés de transparence définies au paragraphe 2.3.2 s'expriment de la manière suivante :

Transparence de localisation : l'accès à une donnée mémoire est réalisé de la même manière et avec la même référence que celle-ci soit stockée localement ou sur un nœud distant.

Indépendance de localisation : le déplacement d'une donnée de la mémoire d'un nœud vers la mémoire un autre nœud ne modifie ni sa référence, ni les méthodes d'accès à cette donnée.

Partage logique : deux processus s'exécutant sur des nœuds distincts peuvent partager des données stockées dans la mémoire de la grappe.

Partage physique : un processus s'exécutant sur un nœud peut utiliser la mémoire physique d'un nœud distant pour y stocker des données.

Il existe de nombreux travaux dans la littérature concernant les mécanismes de gestion globale de la mémoire. On peut cependant regrouper ces travaux en deux catégories. La première concerne les travaux visant à fournir la propriété de partage physique grâce à des mécanismes de **pagination à distance**, tandis que la seconde regroupe les travaux visant à offrir la propriété de partage logique grâce à des mécanismes de **mémoires partagées réparties**. Les propriétés de transparence et d'indépendance à la localisation sont assurée par ces deux mécanismes.

Nous détaillons dans la suite de cette partie les mécanismes de pagination à distance assurant le partage physique de la ressource mémoire et les mécanismes de mémoires partagées réparties assurant le partage logique. Le lecteur non familier avec les mécanismes système de gestion de la mémoire physique peut se reporter à l'annexe A.1 qui présente un rappel des principes de fonctionnement d'un système d'exploitation.

3.1 Partage physique : mécanismes de pagination à distance

La propriété de partage physique permet à un nœud d'utiliser la mémoire physique d'un nœud distant pour stocker des données. Cette propriété peut être réalisée grâce à un mécanisme de **pagination à distance**. Un mécanisme de remplacement classique sur disque offre les propriétés d'indépendance et de transparence à la localisation. Qu'une page soit stockée en mémoire physique ou sur disque n'a aucune incidence sur sa méthode

d'accès ou sa référence. De la même manière, un mécanisme de pagination à distance offre ces deux propriétés : qu'une page soit stockée en mémoire locale ou dans la mémoire d'un nœud distant ne change en rien la méthode d'accès à cette donnée.

Après un rapide rappel de la problématique liée à la pagination à distance, nous présentons dans la suite de ce paragraphe le principe de fonctionnement de ce mécanisme et nous étudions plus particulièrement le système GMS.

3.1.1 Problématique

Le mécanisme de mémoire virtuelle, bien que permettant d'offrir un espace d'adressage de très grande taille, n'est pas une solution acceptable pour les algorithmes manipulant des volumes de données dépassant la quantité de mémoire physique disponible sur un nœud. Le mécanisme de pagination effectuant de nombreux accès disque de petite taille (la taille d'une page), il provoque un effondrement des performances du système. Ce phénomène est également appelé **écroulement** [90].

Le problème de performance dont souffrent les systèmes traditionnels de remplacement sur disque est fortement lié aux performances des disques durs. Même si l'on a assisté à une envolée des bandes passantes offertes par les disques durs (de 2 Mo/s en 1990 à plus de 20 Mo/s aujourd'hui), la latence n'a pas connue une amélioration aussi spectaculaire et reste toujours de l'ordre de quelques millisecondes. De plus, ces performances obtenues dans des conditions optimales (lecture séquentielle d'un grand volume de données) sont loin d'être celles observées dans le cadre d'un système de remplacement. L'accès aléatoire à de petites quantités de données nécessite de nombreux mouvements du bras de lecture, ce qui augmente considérablement la latence et fait chuter brutalement la bande passante.

Les méthodes de programmation dites « *out-of-core* » [16] permettent d'éviter l'écroulement et autorisent l'exécution d'applications dont les données ne peuvent être placées intégralement en mémoire physique. Dans ce cas, l'échange des données entre la mémoire et les disques est assuré par le programmeur. L'objectif est de limiter ces échanges et de maximiser leur taille afin d'utiliser de manière optimale la bande passante des disques. Néanmoins, même si cette méthode offre de bons résultats, elle demande un effort de programmation important, qui augmente de manière significative la complexité intrinsèque des algorithmes à mettre en œuvre.

3.1.2 Principe de la pagination à distance

La **pagination à distance** est une solution alternative pour les applications nécessitant de grandes quantités de mémoire physique. L'idée est de conserver le mécanisme de remplacement de pages d'une mémoire virtuelle, mais d'utiliser la mémoire de nœuds distants comme unité de stockage secondaire. Il est ainsi possible de conserver des méthodes de programmation standard tout en réduisant de manière considérable le surcoût de remplacement.

Définition 9 (Pagination à distance) *La **pagination à distance** consiste à utiliser la mémoire de nœuds distants comme unité de stockage secondaire dans le contexte de la pagination de la mémoire virtuelle.*

Lors d'un remplacement de page, les accès disques sont remplacés par des transferts réseaux. Lorsqu'une page est évincée de la mémoire physique d'un nœud, elle est transférée dans la mémoire d'un nœud distant. Ce mécanisme est appelé **injection**. Les réseaux modernes offrant des latences très faibles et des débit importants, même pour des messages de quelques kilo-octets, le coût d'un remplacement de page est réduit d'un facteur 10 à 20. Le tableau 3.1 présente un comparatif des performances des disques et des réseaux d'interconnexion. Le surcoût logiciel induit par la réception et le stockage d'une page à distance est inclus dans les temps présentés pour l'injection.

Définition 10 (Injection) *Lors d'un remplacement de page, l'**injection** consiste à déplacer le contenu d'un cadre de page vers un cadre de page situé sur un nœud distant.*

	Latence (μs)	Débit crête (Mo/s)	Accès aléatoire à une page de 4 Ko (ms)
Disque	10 000	12	11
Ethernet 100	10	12	1
Myrinet-1	4	126	0.2

TAB. 3.1 – Comparaison des performances d'un disque dur et d'un réseau lors de l'accès aléatoire à une page [38]

Comer et Griffioen [24] ont défini un modèle de gestion de mémoire distante dans lequel des nœuds dédiés font office de serveurs de mémoire. Ces serveurs sont utilisés comme unités de stockage secondaire pour les clients ayant des besoins importants de ressource mémoire. Ce modèle ne propose pas de partage de ressources entre les clients, mais uniquement entre les clients et les serveurs. Felten et Zahorjan [39] ont généralisé ce modèle pour permettre d'utiliser la mémoire de toutes les machines inactives. Lorsqu'une machine devient inactive, le système d'exploitation active un serveur de mémoire qui s'enregistre auprès d'un serveur central. Lorsqu'une machine décide de remplacer une page, elle contacte le serveur central, qui transmet aléatoirement la requête à l'un des serveurs de mémoire enregistrés.

« *Global Memory Management (GMS)* » [37] est un système de gestion globale de la mémoire au sein d'un NOW, qui permet l'injection de pages de n'importe quel nœud vers n'importe quel autre nœud. Contrairement aux systèmes précédents, il ne se contente pas d'injecter les pages de données d'un processus, mais n'importe quelle page du système, y compris les données associées aux systèmes de fichiers. De plus, GMS utilise une gestion globale du remplacement, et non des politiques locales aux différents nœuds du réseau. Enfin, GMS gère automatiquement l'ajout et le retrait de nœuds.

3.1.3 Étude du système GMS

Nous détaillons dans ce paragraphe l'algorithme de remplacement utilisé par le système GMS. Cet algorithme offre une bonne approximation de l'algorithme LRU pour un remplacement de pages à l'échelle de la grappe. Nous présentons ensuite les mécanismes utilisés pour réaliser et mettre en œuvre cet algorithme.

3.1.3.1 Algorithme de remplacement

Au sein d'un réseau utilisant GMS, la mémoire physique des nœuds est divisée dynamiquement en deux parties : la mémoire dite **locale**, contenant des données utilisées par les processus s'exécutant localement et la mémoire dite **globale** contenant des pages provenant d'un remplacement sur une machine distante. De plus, les pages peuvent être soit **privées** soit **partagées**. Les pages partagées peuvent par exemple appartenir à un fichier provenant d'un serveur de fichiers et partagé par plusieurs processus. La mémoire globale ne contient que des pages privées et **propres** (c'est-à-dire ne nécessitant pas une mise à jour sur disque). Lorsqu'un défaut de page se produit sur une machine P, l'algorithme de remplacement de GMS effectue l'une des quatre actions suivantes :

Cas 1 : la page en défaut se trouve en mémoire globale sur un autre nœud, Q. GMS échange la page en défaut de la mémoire globale de Q avec une page quelconque de la mémoire globale de P. La page provenant de Q devient une page locale sur P, augmentant de 1 la taille de la mémoire locale de P. La balance des pages locales et globales de Q n'a pas changée (c.f. figure 3.1).

Cas 2 : la page en défaut se trouve en mémoire globale sur le nœud Q, mais P ne contient que des pages locales. GMS échange la page la plus ancienne[1] de la mémoire locale de P avec la page en défaut de la mémoire globale de Q. La balance des mémoires locales et globales n'est modifiée ni sur P ni sur Q.

Cas 3 : la page en défaut se trouve sur un disque. La page en défaut est chargée dans la mémoire locale de P. GMS choisit la page la plus ancienne du réseau (disons sur le nœud Q) et l'évince de la mémoire de Q après l'avoir écrite sur disque si nécessaire. Enfin, une page de la mémoire globale de P est envoyée dans la mémoire globale de Q. Si P n'a pas de page en mémoire globale, la plus ancienne page de la mémoire locale est envoyée (c.f. figure 3.1).

Cas 4 : la page en défaut est une page partagée située en mémoire locale de Q. Cette page est copiée en mémoire locale de P, tout en laissant l'original sur Q. La page la plus ancienne du réseau (disons sur le nœud R) est évincée après avoir été recopiée sur disque si nécessaire. Une page globale de P est déplacée vers la mémoire globale

[1] Suivant un algorithme LRU.

de R. Si P n'a pas de page en mémoire globale, la plus ancienne page de la mémoire locale est envoyée.

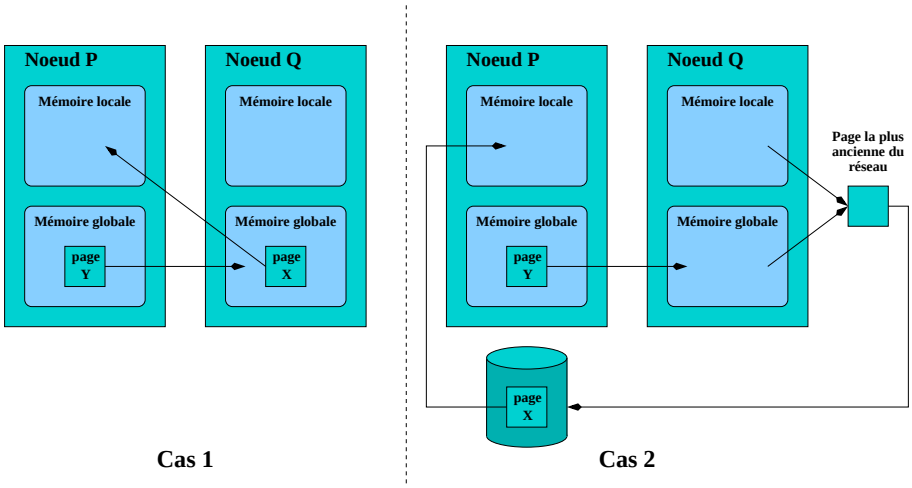

FIG. 3.1 – *Gestion du remplacement de pages dans GMS*

Cet algorithme permet une gestion dynamique des mémoires locales et globales. Lorsqu'un nœud est très actif et consomme beaucoup de mémoire, sa mémoire se remplit de pages locales et il commence à utiliser la mémoire globale de nœuds distants lorsque sa mémoire devient saturée. Les nœuds inactifs hébergeant des pages anciennes (au sens LRU) évincent ces pages afin de stocker des pages globales pour le compte de nœuds distants.

3.1.3.2 Gestion des informations globales

Le mécanisme de remplacement de GMS suppose l'existence d'informations globales sur l'état des nœuds et de leurs pages afin de pouvoir localiser la plus ancienne à l'échelle du réseau. Maintenir une information globale à tout instant étant impossible, chaque nœud ne dispose en fait que d'informations approximatives sur l'état des pages globales. Pour obtenir un compromis acceptable entre la précision des informations globales et le coût de la distribution de ces informations, GMS utilise l'algorithme suivant.

GMS découpe le temps en tranches appelées **époques**. Une époque a une durée maximale T et un nombre maximum de remplacement de pages dans le réseau M. Les valeurs de T et M varient d'une époque à l'autre en fonction de la charge de calcul et de l'état de la mémoire globale. Une nouvelle époque commence lorsque (1) l'époque courante a atteint sa durée maximale T, (2) M pages globales ont été remplacées ou (3) des imprécisions importantes ont été détectées.

Sur chaque nœud, le système conserve des informations sur l'âge des pages locales et globales hébergées sur le nœud. Lorsqu'une nouvelle époque commence, chaque nœud

envoie un résumé de l'âge de ses pages à un **nœud initiateur**. L'initiateur détermine alors l'âge minimum de remplacement $MinAge$, permettant de remplacer M pages dans la prochaine époque. Il calcule également un poids wi associé à chaque nœud i, correspondant au nombre de pages libres ou anciennes (dont l'âge est supérieur à $MinAge$) dont il dispose, proportionnellement aux autres nœuds du réseau. Plus un nœud dispose de pages libres ou anciennes, plus son poids associé est élevé. L'initiateur transmet la liste des poids wi et la valeur $MinAge$ à tous les nœuds et désigne la machine disposant du poids le plus élevé comme prochain nœud initiateur.

Durant une époque, lorsqu'un nœud P doit évincer une page de sa mémoire, il commence par vérifier si l'âge de cette page est supérieur à $MinAge$. Si c'est le cas, la page est simplement détruite (après une éventuelle recopie sur disque). Sinon, P envoie la page au nœud i, où i est choisi aléatoirement avec une probabilité proportionnelle à wi. La page la plus ancienne de i est alors détruite et la page évincée de P devient une page globale sur i.

Cet algorithme permet d'obtenir une approximation de l'algorithme LRU à l'échelle du réseau, puisque si M pages sont détruites globalement durant une époque, ce sont les M plus anciennes du réseau. De plus, il permet de déterminer localement et statiquement lorsque M pages ont été globalement remplacées : lorsque le nœud initiateur a reçu wi pages, il déclare la fin de l'époque en cours.

3.2 Partage logique : mécanismes de mémoire partagée répartie

La propriété de partage logique n'est pas assurée par les mécanismes de pagination à distance. Cette propriété, fondamentale pour un SSI, est offerte par les **mémoires partagées réparties**.

Définition 11 (mémoire partagée répartie) *On appelle **mémoire partagée répartie (MPR)** l'abstraction logicielle ou matérielle d'une mémoire partagée au dessus de modules mémoire physiquement distribués.*

Une mémoire partagée répartie fournit à l'utilisateur l'abstraction d'un espace d'adressage unique au dessus de modules mémoire physiquement distribués. Dans un tel système, des processus s'exécutant sur des nœuds distincts ou non d'une grappe peuvent partager des données stockées dans les différents modules mémoire comme si ces modules ne formaient qu'une mémoire unique et partagée. Les données manipulées par les processus sont placées automatiquement et de manière transparente, à proximité du processeur qui les référence. Une MPR assure donc un modèle de programmation simple et attractif, en vérifiant la propriété de partage logique. Les propriétés de transparence et d'indépendance sont également assurées puisque le programmeur n'a pas à se soucier de la localisation physique des données qu'il référence.

3.2.1 Principe des mémoires partagées réparties

Lorsqu'un processeur désire accéder à un mot mémoire qui n'est pas présent dans sa mémoire locale, la MPR se charge de le rapatrier depuis une mémoire distante disposant d'une copie. Pour tirer profit de la **localité temporelle**, les données rapatriées sont conservées localement dans des caches. L'utilisation de caches réduit certes le nombre d'accès distants et par conséquent le trafic dans le réseau de communication, mais introduit le problème de la cohérence des données dupliquées lorsque celles-ci sont modifiées [20].

Définition 12 (Localité temporelle) *Le concept de **localité temporelle** stipule que des données utilisées récemment ont une forte probabilité d'être à nouveau utilisées dans un futur proche par le même programme.*

Pour rentabiliser la latence du réseau de communication et tirer profit de la **localité spatiale**, les données sont rapatriées par blocs lors de chaque communication. La taille du bloc définit la **granularité** (ou l'unité de partage) sur laquelle la cohérence est maintenue. La taille de ces blocs peut varier de quelques octets à quelques kilo-octets. Le choix d'une taille pour ces blocs influe fortement sur les performances et les mécanismes de détection des accès à ces blocs.

Définition 13 (Localité spatiale) *Le concept de **localité spatiale** stipule que lorsqu'une donnée est utilisée dans un programme, les données placées à des adresses contiguës en mémoire ont une forte probabilité d'être utilisées au cours du même programme.*

Si les MPR ont été mises en œuvre grâce à des composants matériels au sein d'architectures parallèles, de nombreuses mises en œuvre logicielles ont vu le jour depuis les travaux de Kai Li en 1986 [64]. Nous ne nous intéressons dans la suite de ce document qu'aux mises en œuvre logicielles : les **mémoires partagées réparties logicielles** (MPRL).

Définition 14 (mémoire partagée répartie logicielle) *On appelle **mémoire partagée répartie logicielle (MPRL)** la mise en œuvre logicielle d'une MPR.*

Une MPRL permet grâce à des fonctions d'initialisation spécifiques de créer un ou plusieurs espaces d'adressage linéaires accessibles par tous les processus d'une application parallèle ou distribuée (c.f. figure 3.2). Au sein d'une MPRL, les données manipulées par les processus sont identifiées grâce à des adresses virtuelles classiques et sont accédées par les instructions de *lecture* et *écriture* du processeur (respectivement « *load* » et « *store* »). Une application accède donc indifféremment aux données situées en MPRL et aux données situées en mémoire conventionnelle. Un programme écrit pour un multiprocesseur à mémoire partagée n'a donc pas ou peu besoin d'être modifié pour fonctionner sur un système proposant une MPRL.

3.2.2 Modèle mémoire

La réplication des données dans une MPRL pose le problème du maintien de la cohérence des dupliques et de l'ordre des accès aux donnés, c'est-à-dire la manière dont les différents processeurs accèdent aux données. Ce problème est résolu grâce à la définition d'un **modèle mémoire**.

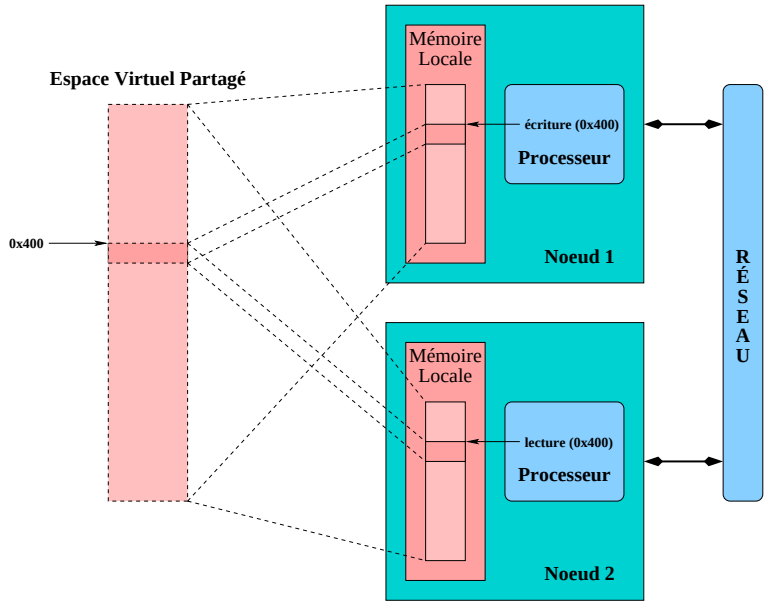

FIG. 3.2 – *Mémoire partagée répartie*

Définition 15 (Modèle mémoire) *Un **modèle mémoire** est un contrat passé entre le programmeur et la MPRL, qui précise le comportement de cette dernière lors des accès mémoire réalisés par une application.*

Le modèle mémoire le plus simple est le modèle réalisant une **cohérence séquentielle** [63] ou **cohérence forte**. Ce modèle assure que (1) toute écriture dans l'espace partagé est immédiatement visible par les autres nœuds et que toute lecture renvoie la dernière écriture, (2) du point de vue de chaque processeur, les opérations d'accès mémoire apparaissent comme étant effectuées dans l'ordre spécifié par le programme. Ce modèle est celui offert par une mémoire physiquement partagée. La mise en œuvre de ce modèle de cohérence sur une architecture à mémoire distribuée peut cependant introduire un surcoût de traitement pouvant être pénalisant en terme de performance.

Les modèles de **cohérence faible** permettent d'augmenter la performance en relâchant l'ordre des programmes et les contraintes d'atomicité des opérations d'écriture. On distingue alors deux types de variables, les variables partagées et les variables de synchronisation qui assurent la protection des accès en écriture sur les variables partagées, ou gèrent la synchronisation entre différents processus [35]. Nous ne traitons pas de ces modèles de cohérence dans ce document. Des informations complémentaires à ce sujet peuvent être trouvées dans [35].

3.2.3 Protocole de cohérence

Un modèle mémoire est mis en œuvre grâce à un **protocole de cohérence**. Un protocole découpe la mémoire en blocs dont la taille peut varier de quelques octets à quelques

kilo octets. A chaque bloc est associé un état qui spécifie les accès autorisés sur l'exemplaire du bloc par le nœud correspondant. Le protocole décrit l'ensemble des opérations à mettre en œuvre lors des accès aux blocs en fonction de l'état associé. Traditionnellement, les protocoles de cohérence sont représentés par des automates d'états finis.

On peut distinguer deux classes principales de protocoles pour la mise en œuvre d'un modèle mémoire à cohérence séquentielle :

- **Les protocoles à diffusion sur écriture** : la modification d'un bloc est immédiatement diffusée à l'ensemble des nœuds disposant d'un exemplaire. Cette classe de protocole issue des MPR matérielles n'est pas envisageable dans une MRPL, le surcoût de diffusion à chaque écriture étant prohibitif dans le cadre d'un mécanisme logiciel. De plus, la latence d'une diffusion dans le cadre d'un NOW ou d'une grappe est beaucoup trop élevée pour que cette diffusion puisse être considérée comme « *immédiate* ».

- **Les protocoles à invalidation sur écriture** : la modification d'une donnée entraîne immédiatement l'invalidation des exemplaires existants de cette donnée. Pour être autorisé à modifier un bloc, un nœud doit jouir d'un accès exclusif à ce bloc. Cette classe de protocole a été très utilisée pour la mise en œuvre de MPRL, comme par exemple dans IVY [64], Shasta [87] ou Blizzard-S [89].

Au sein d'un protocole à invalidation sur écriture, un bloc dispose d'un état global à la grappe et d'un état local à chaque nœud (c.f. figure 3.3). Globalement, un bloc peut être soit dans l'état **lecture**, soit dans l'état **écriture**. Lorsqu'un bloc est dans l'état global *lecture*, il dispose des propriétés suivantes : (1) il peut exister plusieurs copies du bloc dans la grappe et (2) aucun nœud n'est autorisé à écrire sur ce bloc. Lorsqu'un bloc est dans l'état global *écriture*, il dispose des propriétés suivantes : (1) il n'existe qu'un seul exemplaire de ce bloc dans la grappe et (2) le nœud disposant de cette copie peut y accéder en lecture et en écriture. Ce nœud est appelé **propriétaire** du bloc.

Définition 16 (Propriétaire) *On appelle **propriétaire** d'un bloc de données, le nœud disposant, ou étant le dernier à avoir disposé des droits en écriture sur ce bloc. Il est le dépositaire de la copie la plus à jour du bloc.*

En pratique, ce protocole associe à chaque bloc un état local à chaque nœud (c.f. figure 3.3). Cet état reflète localement l'état global du bloc. Trois états sont généralement utilisés :

Invalide : L'accès au bloc est interdit. Cet état permet d'interdire l'accès à un bloc dans l'état global *écriture*, à tous les nœuds ne disposant pas de la copie unique. C'est également l'état de la plupart des blocs à l'initialisation du système.

Lecture : Le bloc ne peut être accédé qu'en lecture. Cet état permet d'autoriser l'accès en lecture à tous les nœuds disposant d'un exemplaire d'un bloc dans l'état global *lecture*.

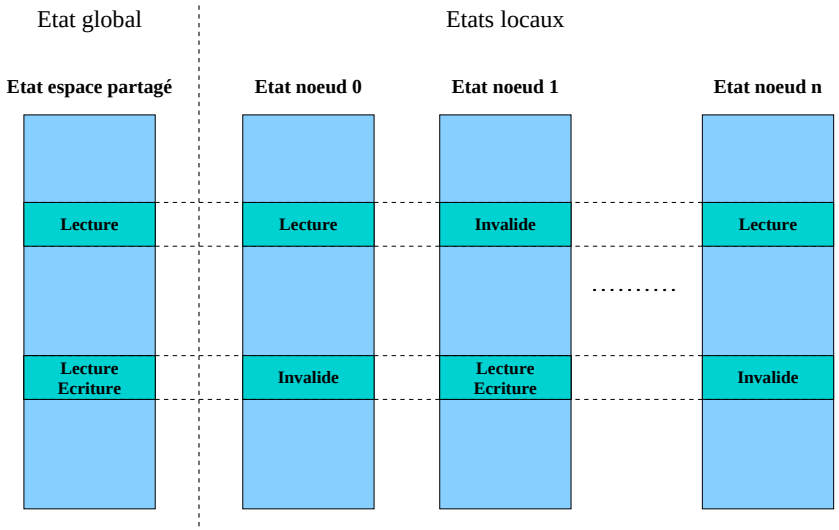

FIG. 3.3 – *État des espaces d'adressage*

Lecture-Ecriture : Le bloc peut être accédé en lecture et en écriture. Cet état permet d'autoriser l'accès en lecture et en écriture au nœud disposant de la copie unique d'un bloc dans l'état global *écriture*. Sur les autres nœuds, ce bloc est impérativement dans l'état *invalide*.

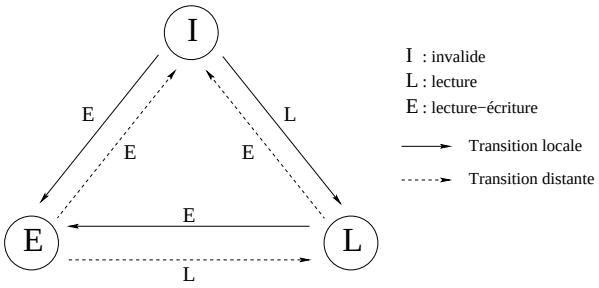

FIG. 3.4 – *Transitions entre les différents états du protocole*

Les changements d'états sont provoqués par les accès aux données partagées par chacun des nœuds. La figure 3.4 présente les transitions possibles entre les trois états du protocole. Les flèches en trait plein indiquent une transition produite par une action locale au nœud. Les flèches en pointillés indiquent que la transition est demandée par un autre nœud. Un bloc passe dans l'état *lecture* sous l'effet d'une lecture, locale ou distante. L'état *lecture-écriture* est atteint lors d'une demande d'écriture locale. Enfin, un bloc passe dans l'état *invalide* lors d'une écriture distante. Le passage d'un état où un bloc est en lecture seule sur plusieurs nœuds à un état où ce bloc est accessible en écriture sur un nœud unique

est appelée phase d'**invalidation**. Elle consiste à supprimer les copies en lecture du bloc avant de donner l'accès en écriture au futur écrivain.

La résolution d'un défaut en écriture sur un bloc X se déroule de la manière suivante :
- Invalidation de toutes les copies ;
- Le propriétaire du bloc X envoie une copie de ce bloc au nœud où s'est produit le défaut puis invalide sa propre copie.
- Le nœud en défaut place son droit d'accès sur le bloc à *lecture-écriture* et devient le nouveau propriétaire.

La résolution d'un défaut en lecture sur un bloc X se déroule de la manière suivante :
- Le propriétaire du bloc X envoie une copie de ce bloc au nœud où s'est produit le défaut puis place son droit d'accès sur le bloc à *lecture*.
- Le nœud en défaut place son droit d'accès sur le bloc à *lecture*.

Ce protocole garantit une cohérence séquentielle. En effet, toute écriture par un nœud sur un bloc interdit l'accès en lecture et écriture pour les autres nœuds. Quand ces derniers ont à nouveau accès au bloc, ils doivent récupérer la dernière version des données et voient donc « *immédiatement* » les opérations d'écritures récentes. La condition d'exécution suivant l'ordre du programme sur un nœud est garantie par le processeur de ce nœud.

3.2.4 Granularité des données partagées

La granularité a une influence importante sur les performances de la gestion mémoire. Pour amortir la latence du réseau de communication et exploiter au mieux la localité spatiale, une granularité importante est un choix judicieux. Toutefois, une granularité importante augmente le risque de **faux partage**. Le faux partage provoque de nombreux déplacements d'un bloc entre les différents nœuds le référençant. Ce phénomène, connu sous le nom de **ping-pong**, surcharge considérablement le réseau et provoque un effondrement des performances.

Définition 17 (Faux partage) *Il y a **faux partage** lorsque plusieurs processus effectuent des accès simultanés (dont au moins l'un d'entre eux est en écriture) à des variables partagées distinctes situées dans un même bloc.*

3.2.4.1 Détournement de la MMU

En général, le choix de la granularité est fixé par la taille de la page mémoire. Parce que la page est l'entité manipulée dans une mémoire virtuelle, le choix de cette dernière comme granularité de maintien de la cohérence offre plusieurs avantages. Tout d'abord, le mécanisme de mémoire virtuelle comporte la notion de droit d'accès. Une combinaison de droits d'accès (lecture, écriture, exécution) est associée à chaque page virtuelle. Ce mécanisme peut être utilisé directement pour représenter les droits d'accès du protocole de cohérence. Ensuite, la MMU détecte automatiquement les violations de droit d'accès aux pages et déclenche des fonctions du système d'exploitation permettant de traiter ces

violations. Ce mécanisme peut donc être très simplement détourné pour détecter les fautes de pages au sens du protocole de cohérence de la MPRL.

La création d'un espace d'adressage partagé est effectuée via la réservation d'une plage d'adresses virtuelles dans l'espace d'adressage des processus. Lorsqu'il y a violation des droits d'accès au sein de cet espace, les mécanismes systèmes sont détournés vers la MPRL qui active le protocole de cohérence. Cependant, ceci implique que la MPRL puisse manipuler la table des pages de chaque processus accédant à l'espace partagé et détourner les mécanismes de résolution de défaut de page du système d'exploitation. Pour cela, il est nécessaire de mettre en œuvre la MPRL dans le noyau du système d'exploitation, comme c'est le cas dans KOAN [61] ou de disposer de mécanismes de niveau utilisateur permettant de modifier la table des pages. Cette dernière solution a été utilisée dans Ivy [64], qui modifie les protections associées aux pages via des appels systèmes et traite les défauts de page grâce à des fonctionnalités offertes au niveau utilisateur par le système d'exploitation.

Bien que le choix d'une granularité de page dans une MPRL offre de nombreux avantages du point de vue de la mise en œuvre, il reste très discutable. En effet, la taille d'une page mémoire n'est pas corrélée à celle des structures de données manipulées par un programme. Il est donc très fréquent que des variables indépendantes soient situées dans la même page, conduisant à des problèmes de faux partage. Ce problème peut être résolu en choisissant une granularité plus faible et plus adaptée aux structures de données manipulées. L'idée est de subdiviser une page en unités de taille plus faible appelées **sous-page** ou **lignes** et de gérer individuellement la cohérence pour chacune de ces unités. Toutefois, le choix d'une granularité plus faible pose de problème de la détection des accès aux blocs. La MMU et les mécanismes matériels généralement disponibles sur les calculateurs ne permettent pas d'associer des droits d'accès ou de détecter les accès sur des zones de mémoire dont la taille est inférieure à la page. Ces fonctions doivent donc être assurées de manière logicielle par la MPRL ou grâce à des astuces de manipulation des pages.

3.2.4.2 Multi-vue

Une solution pour éviter le faux partage serait de placer chaque variable du programme dans une page physique différente. Cette solution conduirait cependant à un important gaspillage de mémoire physique. Une solution intermédiaire est utilisée par le système Millipede grâce au mécanisme de **multi-vues** [53]. Cette solution permet de gérer la cohérence avec une granularité égale à la taille des variables utilisées en détournant les mécanismes de mémoire virtuelle tout en évitant le gaspillage de mémoire physique. Ce mécanisme associe plusieurs pages virtuelles à la même page physique, typiquement une page virtuelle par variable. Considérons trois variables x, y et z dont la taille totale est inférieure à la taille d'une page. Dans une MPRL utilisant une granularité de page, ces trois variables pourrait être allouées au sein de la même page virtuelle, au risque d'introduire du faux partage. Dans un système à multi-vue, chaque variable est allouée dans une page virtuelle différente grâce à une fonction d'allocation spécifique à la MPRL, mais placées dans le même cadre de page à des offsets différents (c.f. figure 3.5). Chaque variable étant placée dans une page différente, il est possible d'associer une protection différente à chaque page

et donc à chaque variable. Le faux partage est ainsi évité.

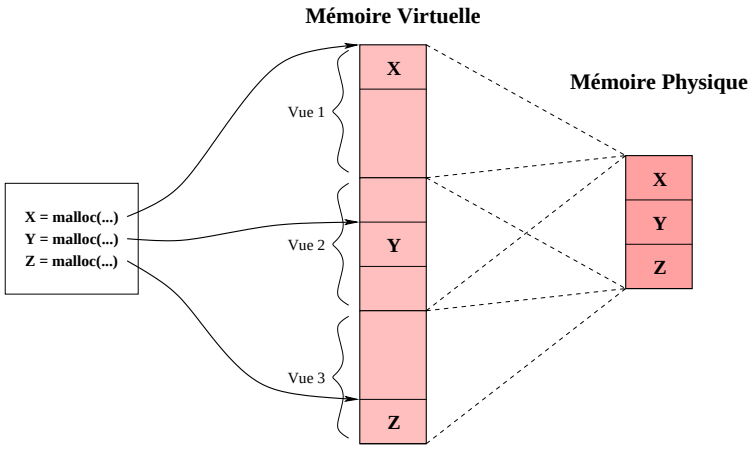

FIG. 3.5 – *Mécanisme de multi-vue*

3.2.4.3 Instrumentation

L'instrumentation d'un programme consiste à identifier statiquement, dans le code assembleur de l'application, les opérations d'accès aux variables partagées et d'enrichir individuellement chacune de ces instructions par des appels à des fonctions qui déterminent l'état de la variable partagée et, si nécessaire, effectuent une opération de cohérence. Cette méthode est utilisée dans les systèmes Midway [12], Shasta [87] et Blizzard-S [89]. L'avantage de cette méthode de détection des accès est que la granularité de cohérence est beaucoup plus fine (32 à 128 octets pour Blizzard-S, 64 ou 128 pour Shasta) que pour les méthodes fondées sur le détournement de la MMU. En revanche, elle nécessite une modification de la chaîne de compilation : l'instrumentation est faite par le compilateur lui-même comme dans Midway, ou sur le fichier binaire produit par le compilateur comme dans Shasta. De plus, comme chaque accès mémoire est instrumenté, le coût induit n'est pas négligeable.

Afin de distinguer les blocs de données partagées des données locales, l'espace d'adressage d'un processus est divisés en **régions**, certaines correspondant aux données partagées, les autres aux données locales. Chaque région de données partagées est divisée en **lignes**, une ligne étant l'unité élémentaire de partage et de gestion de la cohérence. La taille d'une ligne peut varier d'une région à l'autre. L'instrumentation consiste à identifier dans le code original les accès aux régions partagées et y insérer une portion de code prenant l'adresse de chaque accès en paramètre. Ce code vérifie si l'accès à l'adresse considérée est valide au sens du modèle mémoire proposé par la MRPL. Si ce n'est pas le cas, il y a un **défaut de ligne** qui est résolu par le protocole de cohérence.

3.2.4.4 Autres techniques

D'autres méthodes permettent de gérer la cohérence sur des blocs dont la taille n'est pas celle d'une page mémoire. On peut citer par exemple le marquage de régions ou d'objets. Dans le **marquage par régions**, le programmeur regroupe les variables partagées en régions et signale l'accès à ces régions par des appels de fonctions [78, 88]. Avec le **marquage par objets** les fonctions explicites d'accès aux régions partagées sont intégrées dans les méthodes d'accès aux objets [11]. Cette technique a l'avantage d'être transparente pour le programmeur puisque l'appel aux fonctions d'accès aux régions partagées est assurée automatiquement par les mécanismes d'accès aux objets.

3.2.5 Systèmes hybrides

Les réseaux à capacité d'adressage (« *RMA : Remote Memory Access* ») offrent un nouveau modèle de communication en permettant des écritures et/ou des lectures sur un nœud distant de manière directe, sans interruption du processeur distant, donc sans intervention d'aucune couche logicielle (voir chapitre 2.2.1.3). Il est important de noter que ce type de réseau (notamment « *Memory Channel* » et VMMC) ne permet pas de réaliser directement une MPR matérielle car il n'offre pas la cohérence des différentes mémoires locales. En effet ces interfaces de communication sont placées sur le bus d'entrées/sorties de chaque nœud et n'ont donc pas accès au bus mémoire qui relie processeur et mémoire. Elles ne peuvent donc pas observer le trafic mémoire et répercuter une modification locale sur un nœud distant. La technologie SCI permet cependant d'offrir un modèle mémoire proche d'une architecture NUMA, en autorisant la lecture et l'écriture en mémoire distante. Offrir un espace d'adressage partagé au dessus de SCI revient à projeter la mémoire physique d'un nœud dans l'espace d'adressage de plusieurs processus. Cependant, chaque accès à cet espace partagé nécessite une transaction réseau dont le coût n'est pas négligeable (de 2 à 5 microsecondes).

Pour tirer profit des accès distants offerts par ces réseaux tout en limitant le surcoût réseau potentiellement important, des MPR hybrides ont été étudiées. On peut citer Cashmere[59], AURC[51] ou encore SCIOS[58]. Dans SciOS, les segments de mémoire partagée sont présentés comme des fichiers pouvant être projetés dans l'espace d'adressage des processus. Plusieurs types de protocoles sont utilisés pour gérer cet espace partagé. Le plus simple est l'allocation statique de pages physiques : pour chaque page de l'espace virtuel partagé, une page physique est alloué sur un nœud donné et projetée grâce à SCI dans l'espace d'adressage de tous les processus. SciOS offre également un protocole de cohérence relâchée paresseuse. En fonction des accès aux pages, celles-ci sont migrées ou dupliquées. Ainsi, lorsque SciOS détecte un accès en faux partage sur une page, celle-ci est **gelée** sur la machine y effectuant le plus fréquemment des accès et projetée dans l'espace d'adressage des autres nœuds. Le partage est alors géré de façon matérielle. Lorsqu'il n'y a pas de faux partage, les pages sont dupliquées sur les nœuds à la manière d'une MPRL afin d'optimiser la localité temporelle.

Cette solution a l'avantage de tirer partie aussi bien des avantages des nouveaux ré-

seaux de communication que des MPRL. Cependant, elle nécessite l'utilisation de matériel spécifique coûteux, qui limite fortement sa portabilité et sa pérénité.

3.2.6 Éléments de mise en œuvre : étude du système Ivy

Nous présentons dans ce paragraphe quelques éléments concertant la mise en œuvre de la MPRL Ivy [64], qui fait référence dans le domaine des MPRL à cohérence séquentielle.

Dans le système Ivy, les données partagées sont désignées grâce à des adresses virtuelles composée d'un numéro de page virtuelle et d'un déplacement dans cette page. Chaque donnée dispose d'une adresse virtuelle indépendante de la localisation de la page physique la contenant. Ceci assure à la fois la propriété de transparence et d'indépendance à la localisation. La localisation des données est assurée grâce à un mécanisme de répertoire de pages définissant pour chaque page virtuelle de l'espace partagée le ou les nœuds disposant d'une copie de la page.

La mise en œuvre du système Ivy repose sur l'utilisation de deux types d'entités logicielles : les **gestionnaires de pages** qui mettent en œuvre le répertoire de pages et les **serveurs de pages** qui répondent aux requêtes des gestionnaires pour la résolution de défauts de page. Lorsqu'un nœud est en défaut de page, il envoie une requête au gestionnaire de page, qui la transmet au propriétaire. Ce dernier retourne une copie de la page au nœud en défaut.

3.2.6.1 Serveur de pages

Sur chaque nœud est présent un serveur de page, qui remplit deux fonctions. La première est de répondre aux requêtes de demande de pages : lorsqu'il reçoit une demande de page, il détermine l'emplacement de cette page en mémoire et transmet une copie à la machine en défaut. Son deuxième rôle est d'invalider des pages locales sur demande d'un gestionnaire dans le cas d'un accès en écriture distant.

3.2.6.2 Gestionnaire de pages

Le gestionnaire dispose du répertoire de pages contenant pour chaque page le propriétaire de la page et l'ensemble des nœuds disposant d'une copie (*copyset*). Kai Li présente dans sa thèse trois solutions pour mettre en œuvre le gestionnaire :

- La mise en œuvre **centralisée** consiste à utiliser un nœud unique comme nœud gestionnaire. Ce nœud dispose de l'ensemble des informations sur la localisation des propriétaires et reçoit toutes les requêtes de résolution de défauts de pages. Cette mise en œuvre très simple présente l'inconvénient de provoquer un goulot d'étranglement au niveau du nœud gestionnaire, car il peut rapidement être submergé par les requêtes des autres nœuds.
- La mise en œuvre **distribuée statique** consiste à répartir de manière statique la gestion des pages sur l'ensemble des nœuds du système. Une fonction de placement simple (un modulo par exemple) permet d'affecter la gestion d'un sous ensemble de

pages à chaque nœud. La localisation du gestionnaire d'une page donnée est réalisée grâce à cette même fonction. La charge de gestion des pages est ainsi distribuée, ce qui évite l'apparition de goulots d'étranglement.

– Enfin, avec la mise en œuvre **distribuée dynamique** le gestionnaire d'une page n'est plus désigné statiquement. C'est le propriétaire d'une page qui s'occupe de la gestion de celle-ci. Ce propriétaire n'étant pas fixe, sa localisation n'est pas aisée. La solution proposée par Kai Li est d'associer à chaque page sur chaque nœud, un **propriétaire probable** à qui les requêtes de résolution de défauts de page sont transmises. Si le nœud qui reçoit la requête n'est pas ou plus le gestionnaire/propriétaire, la demande est transmise au nœud qui est pour lui le propriétaire probable. La requête est ainsi transmise de propriétaire probable en propriétaire probable jusqu'à arriver au propriétaire effectif. La mise à jour du propriétaire probable est effectuée (1) lorsqu'un nœud reçoit une demande d'accès en écriture à la page, il positionne le propriétaire probable sur le nœud demandeur, (2) lorsqu'un nœud reçoit une demande d'invalidation de page, il positionne le propriétaire probable sur le nœud en défaut ou (3) lorsqu'un nœud obtient les droits en écriture sur une page, il positionne le propriétaire probable sur lui-même.

Le gestionnaire sérialise toutes les requêtes d'accès (lecture et écriture) afin d'assurer l'atomicité des opérations de lecture et d'écriture sur les pages de l'espace partagé. Nous étudions maintenant les mécanismes mis en jeu lors du traitement d'un défaut de page. Nous supposons l'utilisation d'un gestionnaire centralisé ou distribué statiquement.

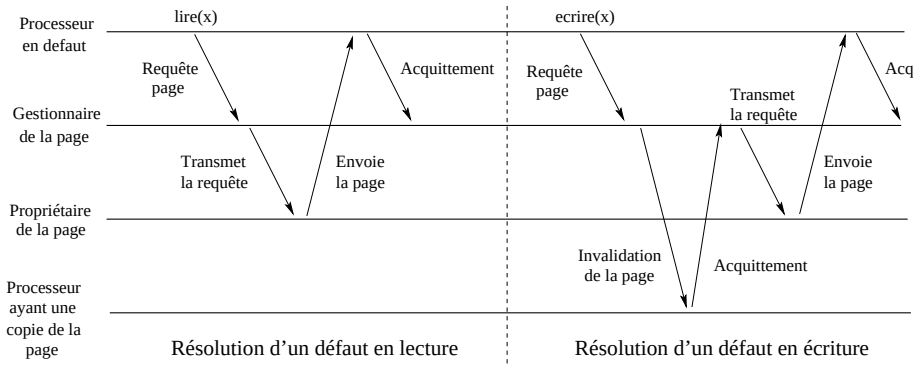

FIG. 3.6 – *Résolution d'un défaut de page*

3.2.6.3 Traitement d'un défaut en lecture

Lorsqu'un défaut de page en lecture se produit sur un nœud, le gestionnaire de défauts de la MPRL envoie une requête de lecture pour la page p au nœud gestionnaire (c.f. figure 3.6). Le gestionnaire ajoute le nœud en défaut à la liste des nœuds disposant d'une copie de la page p et transmet la requête vers le nœud propriétaire. Le propriétaire envoie alors une copie de la page au nœud en défaut et place le droit d'accès à sa propre copie

en lecture seule. Enfin, le nœud en défaut reçoit la copie, place son droit d'accès sur la page p à lecture seule et envoie un acquittement au gestionnaire, indiquant ainsi la fin du traitement du défaut de page.

3.2.6.4 Résolution d'un défaut en écriture

Lorsqu'un défaut de page en écriture se produit sur un nœud, le gestionnaire de défauts de la MPRL envoie une requête d'écriture pour la page p au nœud gestionnaire (c.f. figure 3.6). Le gestionnaire envoie des requêtes d'invalidation de la page p à tous les nœuds disposant d'une copies et attend les acquittements. Ensuite, il transmet la requête au propriétaire de la page. Celui-ci envoie une copie de la page au nœud en défaut et invalide sa copie. Enfin, le nœud en défaut reçoit la copie, place son droit d'accès sur la page p à lecture-écriture et envoie un acquittement au gestionnaire. A la réception de ce message, le gestionnaire note que le nœud en défaut est devenu le nouveau propriétaire.

3.3 Discussion

Nous avons présenté dans ce chapitre différents mécanismes logiciels permettant de réaliser les propriétés de transparence. Ces mécanismes peuvent être regroupés en deux catégories : les mécanismes de pagination à distance assurant la propriété de partage physique et les mécanisme de mémoire partagée répartie logicielle assurant la propriété de partage logique. Ces deux types de mécanisme assurent également les propriétés de transparence et d'indépendance à la localisation. Le tableau 3.2 présente un résumé des propriétés offertes par différents systèmes mettant en œuvre ces mécanismes.

Dans le cas de la pagination à distance, l'objectif est d'utiliser au mieux les ressources disponibles à travers le réseau. La pagination à distance permet de placer dans la mémoire d'un nœud sous-utilisé les données d'une application s'exécutant sur un autre nœud, lorsque la mémoire de celui-ci est saturée. L'application a l'illusion que le nœud sur lequel elle s'exécute dispose d'une grande quantité de mémoire physique. L'effet obtenu est le même qu'avec la pagination sur disque, mais en utilisant le réseau et la mémoire de nœuds distants comme unité de stockage secondaire, les performances du système sont améliorées de façon spectaculaire. Cependant, ce mécanisme ne masque aucunement la multiplicité des nœuds et de manière incomplète la distribution des modules mémoire. Un processus peut utiliser la mémoire d'un nœud distant de manière transparente, mais il ne peut partager cette mémoire avec un autre processus situé sur un autre nœud. Ce mécanisme assure donc la propriété de transparence et d'indépendance à la localisation ainsi que le partage physique, mais pas la propriété de partage logique.

La propriété de partage logique est assurée par les mémoires partagées réparties. Une MPRL permet à des processus d'une application parallèle s'exécutant sur des nœuds distincts de partager des variables situées en mémoire. Pour cela, les données référencées par un processeur sont automatiquement migrées ou dupliquées dans sa mémoire locale. La duplication des données pose des problèmes de cohérence résolus grâce à des protocoles de gestion de la cohérence. Les différentes mises en œuvre de MPRL existantes souffrent de

nombreuses limitations. À l'exception de quelques systèmes comme par exemple la MPRL Koan [61], rares sont les MPRL assurant la propriété de partage physique. De plus, la pagination sur disque local est rarement supportée, ce qui implique que la taille des données manipulées sur chaque nœud ne doit pas excéder la quantité de mémoire disponible localement. Les mises en œuvre de niveau utilisateur nécessitent l'ajout de fonctions d'initialisation et d'allocation mémoire particulières dans le code du programme et supportent rarement le partage de mémoire entre différentes applications parallèles. Les mises en œuvre dans le noyau d'un système d'exploitation nécessitent une modification lourde du noyau, qui implique souvent de nombreuses restrictions : système mono-utilisateur, pas de gestion du remplacement, interaction limitée avec le système de fichier, etc.

Enfin, ces systèmes proposent uniquement une gestion globale de la mémoire. La gestion des processeurs (placement et ordonnancement des processus) et des disques (accès aux disques distants, accès haute performance) est laissée à la charge du programmeur. Les objectifs d'un SSI bien qu'atteints pour la gestion mémoire, ne le sont pas pour l'ensemble des ressources de la grappe. La vision d'une machine unique est donc très imparfaite.

	Niveau de mise en œuvre	Transparence à l'utilisation	Transparence de localisation	Indépendance de localisation	Partage physique	Partage logique
GMS	Noyau	Oui	Oui	Oui	Oui	Non
Ivy	Utilisateur	Non[a]	Oui	Oui	Inc[1]	Oui
Koan	Noyau	Non[a]	Oui	Oui	Oui[2]	Oui
Shasta	Noyau	Non[b]	Oui	Oui	Inc[1]	Oui
Blizzard-S	Noyau	Non[b]	Oui	Oui	Inc[1]	Oui
SciOS	Noyau/matériel	Non[a]	Oui	Oui	Oui	Oui

[a] L'utilisation de ce système nécessite l'insertion de primitives spécifiques dans les applications, notamment lors de l'initialisation de la MPRL et de la création de région de mémoire partagées.
[b] L'application doit être recompilée ou instrumentée afin de permettre le fonctionnement de la MPRL.
[1] Les données d'une application peuvent être placées dans la mémoire d'un nœud distant pour satisfaire le protocole de cohérence. Il n'existe cependant pas de mécanisme d'injection.
[2] Pas de remplacement sur disque.

TAB. 3.2 – Comparaison des propriétés offertes par différents systèmes de gestion globale de la mémoire

4 GESTION GLOBALE DE LA RESSOURCE DISQUE

Une grappe dispose généralement d'un grand nombre de disques, typiquement un disque par nœud. Cependant, seul le disque attaché au nœud où s'exécute un processus peut être vu et accédé par ce processus[1]. Or, il serait très avantageux de pouvoir utiliser les disques situés sur les nœuds distants, afin d'augmenter la capacité de stockage, de répartir la charge d'accès aux disques, voire d'augmenter la bande passante lors de l'accès aux fichiers.

A cet effet, il est possible de définir un SSI fondé sur une gestion globale des disques pour lequel les propriétés de transparence définies au paragraphe 2.3.2 s'expriment de la manière suivante :

Transparence de localisation : un processus accède de la même manière aux fichiers qu'ils soient stockés sur son nœud d'exécution ou sur un nœud distant.

Indépendance de localisation : le déplacement d'un fichier d'un nœud de stockage vers un autre ne modifie ni les méthodes d'accès à ce fichier, ni sa référence externe.

Partage logique : un fichier peut être ouvert simultanément par plusieurs processus s'exécutant sur des nœuds distincts.

Partage physique : un fichier créé par un processus s'exécutant sur un nœud peut être stocké sur le disque d'un autre nœud.

Deux types de mécanismes ont été proposés afin offrir un SSI fondé sur une gestion globale des disques. La première approche consiste à permettre à un nœud d'accéder au disque d'un nœud distant grâce à un **Système de Gestion de Fichiers Distribué (SGFD)**. De nombreux systèmes ont été mis en œuvre et sont largement utilisés. On peut citer NFS [84], CODA [86], AFS [68] ou encore xFS [7].

La seconde approche provient du monde des calculateurs parallèles. Elle consiste à fragmenter un fichier et à stocker les fragments sur plusieurs disques grâce à un **Système de Gestion de Fichiers Parallèle (SGFP)**, tel que Vesta [26], xFS [7], PPFS [50] ou Galley [73]. Cette solution permet d'accéder en parallèle à un ensemble de disques pour la lecture ou l'écriture d'un fichier, augmentant ainsi la bande passante totale et la capacité de stockage.

Afin d'optimiser les performances d'accès aux disques, les SGFD et SFGP utilisent des caches présents sur les différents nœuds de l'architecture. Ces caches distribués peuvent être considérés comme une ressource à part entière, qu'il est possible de partager à travers la grappe grâce à des mécanismes de **caches coopératifs** [30].

[1] En l'absence de système de fichiers distribué tel que NFS par exemple

Après avoir donné une définition d'un système de gestion de fichiers distribué et d'un système de gestion de fichiers parallèle, nous détaillons dans la suite de ce chapitre comment ces systèmes réalisent les propriétés de transparence et d'indépendance à la localisation ainsi que les propriétés de partage logique et physique. Enfin, nous présentons les mécanismes de caches de fichiers coopératifs assurant une gestion globale des caches de fichiers. Le lecteur non familier avec les mécanismes système de gestion des disques peut se reporter à l'annexe A.2 qui présente un rappel des principes de fonctionnement d'un système d'exploitation.

4.1 Définition

Nous donnons dans ce paragraphe une définition des systèmes de gestion de fichiers distribué et parallèle et mettons en évidence les problèmes de conception liés à ces mécanismes.

4.1.1 Système de gestion de fichiers distribué

Dans une architecture distribuée, chaque nœud dispose d'un ensemble d'unités de stockage pouvant héberger des fichiers. Dans cet environnement, les SGFD sont utilisés afin de permettre à un processus s'exécutant sur une machine A d'accéder à un fichier stocké sur une machine B.

Définition 18 (Système de gestion de fichiers distribué) *Un **système de gestion de fichiers distribué (SGFD)** est un service système permettant un accès transparent via une interface unique et standard, aussi bien aux disques locaux d'un nœud qu'aux disques de nœuds distants.*

Un SGFD est architecturé autour d'un ensemble de clients et de serveurs (voir figure 4.1). Chaque serveur s'exécute sur sa propre machine et met en œuvre les fonctionnalités d'un SGF centralisé pour son disque local. Les clients s'exécutent sur les machines des utilisateurs et font le lien entre les applications et les serveurs distants. Sur certains systèmes, un nœud peut être à la fois client et serveur. Afin d'optimiser les performances, les nœuds serveurs utilisent des caches permettant de stocker des fichiers provenant des disques locaux et donc de limiter le nombre d'accès réels aux disques. Les nœuds clients utilisent des caches permettant de stocker des fichiers provenant des serveurs afin de limiter les transactions réseau.

Au sein d'un SGFD, l'accès à des fichiers stockés sur différents nœuds par des processus s'exécutant sur de multiples nœuds pose de nombreux problèmes :

Désignation : un SGFD doit offrir un système de désignation permettant de désigner de manière unique un fichier donné à travers un ensemble de nœuds.

Localisation : un utilisateur désigne un fichier grâce à un **nom externe**, typiquement une chaîne de caractère. Un SGFD doit être capable de retrouver le nœud hébergeant le fichier grâce à ce nom.

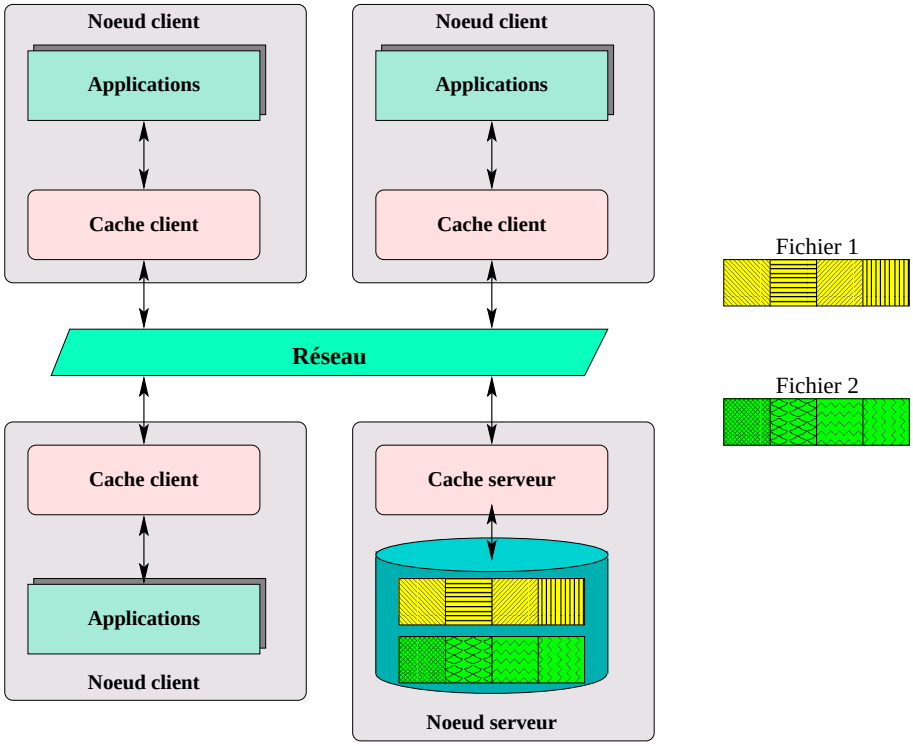

FIG. 4.1 – *Architecture d'un SGFD et stockage physique des données*

Transparence : idéalement, pour qu'un SGFD corresponde à la définition d'un SSI, il doit offrir une transparence totale vis-à-vis de la localisation des fichiers. L'utilisateur ne doit pas avoir à se soucier de la localisation exacte d'un fichier pour pouvoir y accéder.

Cohérence : au sein d'un SGFD, un même fichier peut être accédé simultanément par plusieurs processus situés sur des nœuds différents. Plusieurs copies d'un même fichier peuvent donc exister et être modifiées simultanément, posant des problèmes de cohérence.

4.1.2 Système de gestion de fichiers parallèle

4.1.2.1 Problématique

Les SGFP ont été conçus à l'origine pour tenter de combler le gouffre qui sépare les performances des unités de stockage et la grande capacité de traitement des machines. En effet, en 10 ans la puissance des microprocesseurs a été multipliée par 50 alors que la technologie des disques a peu évolué (voir tableau 4.1). De plus, l'utilisation d'une grande quantité de processeurs au sein d'architectures distribuées permet d'obtenir des puissances de calcul encore plus élevées, amplifiant le problème de manière considérable.

Une autre limitation des disques actuels est leur capacité (quelques dizaines de gigaoctets). Bien que suffisante pour les applications traditionnelles, elle devient limitée pour

Paramètre matériel	Amélioration / an	1990	2001
Latence disque	10%	10 ms	3 ms
Bande passante disque	20%	3 Mo/s	20 Mo/s
Puissance CPU	50%	20 Megaflops	1000 Megaflops

TAB. 4.1 – Amélioration des performances par année de différents matériels [28]

les applications scientifiques, qui peuvent utiliser des fichiers dont la taille avoisine fréquemment le téra-octet.

Les SGFP offrent une solution permettant d'augmenter considérablement l'espace de stockage et la bande passante d'accès aux fichiers, tout en masquant la multiplicité et la distribution des disques.

4.1.2.2 Définition d'un SGFP

En confinant un fichier sur une unité de stockage unique, on limite la bande passante à celle de l'unité. En distribuant ce fichier sur plusieurs disques, on peut y accéder en parallèle et ainsi augmenter potentiellement la bande passante d'un facteur égal au nombre de disques. C'est sur ce principe que sont bâtis les systèmes de gestion de fichiers parallèles.

Définition 19 (Système de gestion de fichiers parallèles) *Un **système de gestion de fichiers parallèles** (SGFP) est un service système permettant de gérer l'accès en parallèle à des fichiers dont les données sont physiquement réparties sur un ensemble de disques répartis au sein d'une architecture distribuée.*

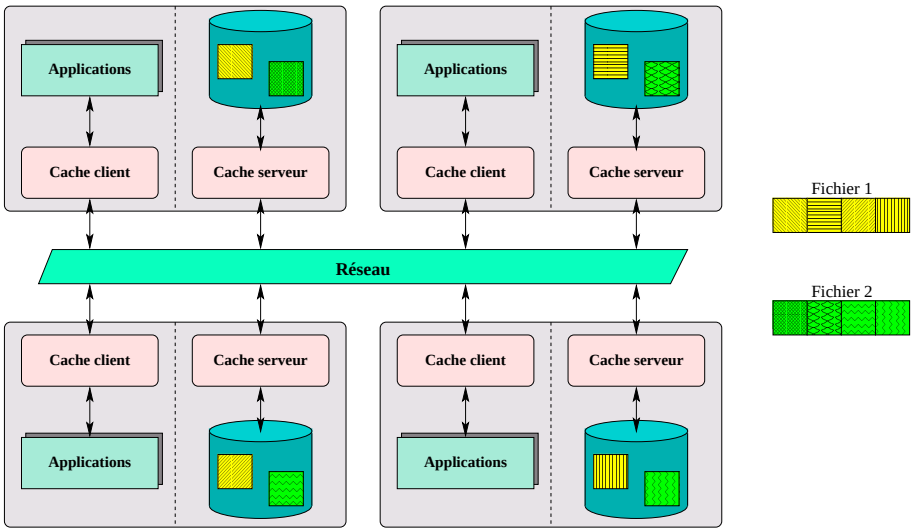

FIG. 4.2 – *Architecture d'un SGFP et stockage physique des données*

Au sein d'un SGFP, les fichiers sont découpés en **fragments**, de taille variable suivant les systèmes. Ces fragments sont distribués sur les disques et peuvent être accédés en parallèle (c.f. figure 4.2). Un SGFP peut être vu comme une mise en œuvre logicielle d'un système RAID [77] au dessus d'un ensemble de disques distribués. Le transfert de données entre une mémoire et un ensemble de disques n'est pas assurée par un contrôleur matériel via un DMA, mais par un ensemble de serveurs communicants via un réseau. Les problèmes posés par cette architecture sont nombreux et liés à la distribution de la gestion des données et au coût des transferts réseaux :

Distribution : un SGFP doit utiliser une répartition des fragments permettant de réaliser un maximum d'accès aux disques en parallèle quelque soit la requête. Une mauvaise répartition pouvant conduire à des accès séquentiels.

Localisation : un SGFP doit pouvoir retrouver rapidement les fragments d'un fichier sans encombrer inutilement le réseau de communication. La structure standard d'i-nœud utilisée dans UNIX ne suffit pas. La gestion d'informations complémentaires est nécessaire afin de localiser l'ensemble des fragments d'un fichier sur les différents disques.

Cohérence : l'accès simultané à un même fichier depuis les différents processus d'une application parallèle localisés sur différents nœuds pose des problèmes de cohérence et de gestion des accès, notamment au niveau de la sémantique du pointeur de fichier et de la cohérence des données dupliquées.

4.2 Transparence et indépendance à la localisation

La gestion de la transparence et de l'indépendance à la localisation des fichiers est directement liée aux problèmes de désignation et de localisation. La méthode d'accès à un fichier (son nom externe et son interface d'accès) doit être indépendante de la localisation du fichier dans l'architecture distribuée. Nous présentons dans la suite de ce paragraphe un bref aperçu des travaux visant à remplir les objectifs de transparence et d'indépendance à la localisation.

4.2.1 Problème de désignation

Au sein d'un SGFP, un fichier n'est pas stocké sur un nœud particulier d'une architecture, mais fragmenté sur les disques de tous les nœuds de cette architecture. Ainsi le problème de désignation n'apparaît pas puisqu'il n'est pas nécessaire de faire de lien entre un nom de fichier et le nœud hébergeant ce fichier. Certains SGFP autorisent la fragmentation d'un fichier sur un sous-ensemble des nœuds, posant alors des problèmes de désignations. Ces problèmes sont cependant similaires à ceux rencontrés dans les SGFD, c'est pourquoi nous nous limitons dans ce paragraphe à l'étude du problème de désignation dans le cadre des SGFD.

Le système de désignation est très important dans un SGFD puisqu'il influe directement sur la localisation d'un fichier et la transparence des accès distants. Pour illustrer les

différentes méthodes de désignation que nous présentons, nous supposons l'existence d'une machine nommée *air* hébergeant un serveur de fichiers. Sur ce serveur est stocké un fichier nommé *these.tex* situé dans le répertoire *rlottiau*.

Le mécanisme de désignation le plus simple consiste à associer au nom de chaque fichier, le nom du serveur l'hébergeant. L'accès à notre fichier exemple serait donc le suivant : */air/rlottiau/these.tex*. Ce mécanisme très simple, notamment utilisé dans le système **Ibis** [92], est totalement inadapté à la conception d'un SSI puisqu'il n'offre ni transparence, ni indépendance à la localisation. Le chemin d'accès à un fichier est dépendant du nom de la machine l'hébergeant. Une solution plus élégante a été largement popularisée grâce au système de fichier NFS de Sun.

NFS utilise un système de montage permettant d'abstraire la localisation physique d'un fichier. Le protocole de montage permet d'établir une connexion logique initiale entre un client et un serveur. Cette connexion consiste à rendre accessible de manière transparente un répertoire d'une machine serveur à une machine distante. Cette connexion est établie sur un nœud client grâce à une opération de montage. La sémantique de l'opération est la suivante : le répertoire éloigné est monté sur un répertoire du SGF local à la machine cliente. Une fois l'opération de montage terminée, le répertoire monté présente l'allure d'une sous-arborescence intégrale au SGF local, en remplaçant la sous-arborescence descendant du répertoire local (voir figure 4.3). Notre fichier exemple est donc accessible grâce au nom */mnt/rlottiau/these.tex*. Ce chemin est transparent à la localisation puisqu'il ne dépend pas du nom du nœud hébergeant le fichier, néanmoins si le fichier est déplacé vers un autre serveur, son chemin d'accès n'est plus valide. Cette méthode de désignation n'est donc pas indépendante à la localisation.

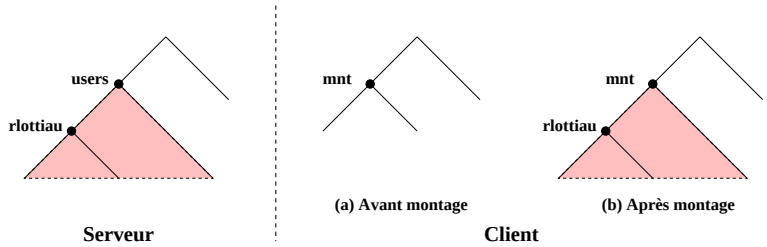

FIG. 4.3 – *Montage de répertoires distants dans NFS*

Le système de fichier Andrew [68] (AFS) développé à l'université Carnegie Mellon distingue deux espaces de désignation : l'**espace local de nom** et l'**espace partagé de nom**. L'espace local de nom est le système de fichier racine d'une station de travail, à travers lequel l'espace global de nom est accessible via le répertoire spécialisé **/cmu**. L'espace local de nom est différent pour chaque station de travail et contient les programmes systèmes essentiels pour un fonctionnement autonome et une meilleure performance, ainsi que les fichiers temporaires. Un réseau de machines utilisant Andrew est découpé en sous-réseaux composés d'un ensemble de stations de travail et d'un serveur de fichier assurant le stockage des fichiers de l'espace partagé de nom (voir figure 4.4). Les serveurs peuvent

communiquer pour accéder à un fichier partagé stocké sur un serveur distant. Avec Andrew, notre fichier exemple aura pour nom */rlottiau/these.tex* s'il est local à la station de travail et */cmu/rlottiau/these.tex* s'il est global et partagé. Le chemin d'accès à un fichier dans l'espace global de nom est transparent et indépendant à la localisation. Il subsiste cependant une différence de désignation entre un fichier local et global, introduisant une légère imperfection dans la transparence de localisation. Cette imperfection est résolue par les systèmes Sprite [76], Locus [81] ou xFS [7], qui ne font aucune distinction entre un fichier local et distant. Notre fichier exemple dispose donc d'un nom unique, quelque soit sa localisation : */rlottiau/these.tex*.

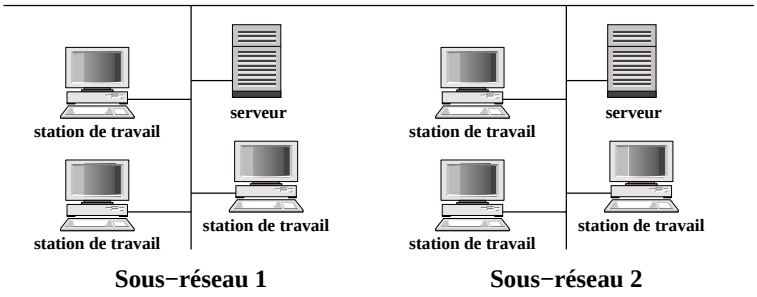

FIG. 4.4 – *Architecture d'un réseau Andrew*

4.2.2 Problème de localisation

Au sein d'un SGFP, le problème n'est pas de localiser un fichier en fonction de son nom, mais de localiser les blocs constituants ce fichier. La gestion de la localisation des blocs d'un fichier est réalisée grâce à une structure comparable à une structure d'i-nœud composée de tuples de la forme *{nœud, contrôleur, disque}*, qui permettent de définir pour chaque bloc, le nœud, le contrôleur et le disque hébergeant ce bloc [18]. Ces données sont stockées dans des métas fichiers associés aux fichiers de données et sont dupliquées sur les différents disques afin d'éviter les problèmes de contention. Dans certain systèmes comme PIOUS[69] ou ParFiSys[19] la distribution des blocs est cyclique et statique (voir paragraphe 4.4.2), ce qui permet de déterminer la localisation d'un bloc par un simple calcul, sans nécessiter l'utilisation de méta-données.

Au sein d'un SGFD, le problème de la localisation consiste à déterminer sur quel nœud est stocké un fichier en fonction de son nom. Cette localisation est étroitement liée au mécanisme de désignation utilisé. Plus la désignation est transparente et indépendante à la localisation, plus le mécanisme de localisation est complexe.

La localisation d'un fichier dans un SGFD tel Ibis, adoptant une désignation de type *machine:nom-local* est triviale. Le fichier nommé *machine:nom-local* est stocké sur le nœud *machine* sous le nom *nom-local* dans le SGF de ce nœud.

Au sein du système NFS, la localisation d'un fichier est plus complexe. L'accès aux fichiers distants est réalisé grâce au mécanisme de montage qui permet de placer dans

l'arborescence locale d'un SGF un répertoire distant et ainsi d'accéder aux fichiers qu'il contient. Le mécanisme de localisation n'est pas utilisé lors de l'accès à chaque fichier, mais uniquement lors de la traversée d'un répertoire monté. Le noyau dispose d'une table contenant les informations de montage telles que le point de montage, le nom du répertoire monté et le nom du serveur de fichier hébergeant le répertoire monté. Si notre fichier exemple est stocké dans le répertoire */users/rlottiau* sur le serveur *air* (c.f. figure 4.3) et que le répertoire */users* est monté localement sur */mnt*, le noyau dispose d'une entrée dans la table de montage indiquant que le répertoire local */mnt* est un point de montage pour le répertoire */users* hébergé sur la machine *air*. Grâce au nom de fichier et à la table de montage, NFS est en mesure de localiser tous les fichiers de l'arborescence.

Dans le système Andrew, la localisation d'un fichier est assurée par les serveurs des sous-réseaux. L'espace partagé de nom est constitué d'unités appelées **volumes**, regroupant généralement les fichiers d'un client unique. Les volumes sont groupés par un mécanisme similaire à celui du montage de UNIX. L'information sur l'emplacement est stockée dans une **base de données d'emplacements de volumes** entièrement dupliquée sur chaque serveur. Un client peut identifier l'emplacement de chaque volume dans le système en consultant cette base de données. Afin d'équilibrer l'espace disque disponible et l'utilisation des serveurs, les volumes peuvent migrer entre les partitions disques et les serveurs. Lors d'une migration, le serveur d'origine continue à servir les requêtes jusqu'à la terminaison de la migration. Une fois la migration terminée, la base de donnée d'emplacement des volumes est mise à jour pour activer le nouveau serveur.

4.2.3 Accès aux données

Au sein d'un SGFD, les fichiers sont accédés grâce à l'interface standard d'accès aux fichiers composée des fonctions *ouverture/déplacement/lecture/écriture/fermeture*. Cette interface est la même que le fichier soit local ou distant. La transparence est donc assurée au niveau de l'interface.

Cette transparence n'est cependant pas assurée par les SGFP. Pour des raisons de performance, des interfaces d'accès spécifiques sont généralement utilisées pour manipuler les fichiers parallèles. Des études sur les accès aux fichiers dans les applications parallèles ont mis en évidence une certaine régularité [66, 74]. De nombreuses applications effectuent des accès par **pas**. Un accès par pas consiste à accéder successivement à un ensemble de blocs de données de taille fixe N espacées de P octets.

Une interface d'accès standard ne permet pas d'effectuer ce genre d'accès facilement et efficacement. C'est pourquoi de nombreuses interfaces spécialisées ont été proposées afin d'offrir des modèles d'accès aux fichiers plus proches des schémas de programmation des programmes parallèles. De plus, en connaissant le schéma d'accès aux données, les SGFP peuvent effectuer plus facilement la correspondance entre les accès aux fichiers et la distribution des données sur les disques. Il est ainsi possible de grouper les requêtes d'accès aux fichiers, afin de pouvoir optimiser le pré-chargement et l'ordonnancement des accès physiques, c'est-à-dire l'ordre dans lequel les blocs sont lus. Un bon ordonnancement doit minimiser les mouvements du bras de lecture afin d'optimiser la banse passante, tout en

assurant un temps de réponse faible pour chacune des requêtes disque.

Le système Vesta [26] introduit une interface utilisant la notion de fichier à deux dimensions. Un fichier est découpé en **cellules**, elles-mêmes découpées en **registres**. Les cellules sont des portions contiguës d'un fichier définies à la création du fichier. Les registres sont les unités élémentaires de distribution de données. Cette division en cellules et registres permet à l'utilisateur de voir un fichier comme un tableau à deux dimensions.

Le système Galley [73] propose une série d'interfaces génériques permettant de réaliser des schémas d'accès complexes :

Accès par pas : Permet de lire n segments de m octets espacés de p octets.

Vecteur de pas : Il s'agit d'une extension du premier mode qui permet de préciser la valeur de p entre chaque segment.

Vecteur d'accès : Permet de préciser pour chaque accès l'offset du début du segment et sa longueur [60]. Cette technique permet de grouper des requêtes, même très différentes, au sein d'une seule requête.

4.3 Partage logique des fichiers

Un SGFD permet à plusieurs processus s'exécutant sur des nœuds distincts d'accéder simultanément à un même fichier. L'accès distribué à un fichier pose cependant des problèmes de cohérence des données, notamment à cause de la présence de caches de fichiers sur les nœuds clients. Lorsqu'un nœud A écrit dans un fichier, les données correspondantes sont placées dans son cache local. Si un nœud B effectue ensuite une lecture sur ce même fichier, il n'a pas connaissance des modifications effectuées par le nœud A.

Le problème de la cohérence des accès distribués consiste à trouver un compromis acceptable entre une mise en œuvre simple mais conduisant à une incohérence totale des accès distribués et l'introduction de mécanismes complexes et coûteux de gestion de la cohérence des caches clients. Ce compromis a des répercutions sur la gestion des caches et sur la sémantique des accès aux fichiers fournie à l'utilisateur.

4.3.1 Sémantique de partage

Dans un système de gestion de fichiers centralisé, lorsqu'une opération de lecture sur un fichier suit une opération d'écriture, l'opération de lecture retourne la valeur de la dernière écriture. Ce modèle d'accès aux fichiers partagés est connu sous le nom de **sémantique UNIX**[90]. Cette sémantique est cependant très difficile à mettre en œuvre dans un système distribué, notamment à cause du problème de gestion de la cohérence des caches clients.

D'autres sémantiques permettent une mise en œuvre plus légère de la cohérence des caches, en relâchant la cohérence des accès. C'est notamment le cas de la **sémantique de session**. La sémantique de session stipule que toutes les modifications apportées à un fichier ne sont visibles par les autres processus qu'après la fermeture du fichier. Si plusieurs processus modifient un même fichier simultanément, le contenu final de ce fichier sera égal au contenu du fichier ayant été fermé le dernier.

On peut citer d'autres sémantiques, telles que la **sémantique de fichiers immuables**, qui précise qu'un fichier peut être créé, lu, mais pas modifié, supprimant ainsi tout problème de cohérence des écritures distribuées.

Enfin, la **sémantique de transaction** assure qu'entre le début et la fin d'une transaction, toutes les opérations d'accès à un fichier sont atomiques et que le résultat de l'exécution de transactions concurrentes est équivalent à l'exécution séquentielle (dans un ordre non défini) de ces transactions.

4.3.2 Impact sur la gestion de la cohérence des caches

La sémantique de partage est intimement liée aux algorithmes et à la granularité de gestion de la cohérence des caches de fichiers distribués. Nous nous attachons ici à présenter les mécanismes utilisés par différents SGFD pour le maintien de la cohérence des caches de fichiers distribués.

4.3.2.1 Cache inactif en écriture

La première solution, qui est également la plus radicale, consiste à interdire de placer dans le cache un fichier partagé ouvert en écriture. Ainsi, aucun problème de cohérence ne peut apparaître. Cependant, tous les accès en lecture et écriture étant sérialisés vers le serveur en court-circuitant le cache local, les performances sont très inférieures à celles obtenues dans d'autres systèmes distribués. Cette approche est notamment utilisée dans le système Sprite [72].

4.3.2.2 Vérification sur accès

Avec le protocole de vérification sur accès, chaque bloc dispose d'une copie maître située sur le serveur et maintenue à jour en permanence. Pour cela, chaque écriture dans un cache local est transmise aussitôt au serveur afin de mettre à jour la copie maître. Lors de chaque accès en lecture à un bloc situé dans le cache local, le SGFD vérifie auprès du serveur que la copie locale est cohérente avec la copie maître. Si ce n'est pas le cas, la copie locale est remise à jour.

Ce protocole est très coûteux en terme de consommation de bande passante et peu efficace puisqu'il nécessite de nombreuses communications réseaux. Pour améliorer les performances, il est possible de ne vérifier la cohérence des copies locales que périodiquement. Mais dans ce cas, les caches peuvent être temporairement incohérents. C'est la solution retenue par NFS [84], qui ne vérifie la validité des blocs que s'ils sont dans le cache depuis plus de trois secondes.

4.3.2.3 Copie unique

Dans le système PAFS [27], chaque bloc d'un fichier donné dispose d'un emplacement unique et fixe dans l'ensemble des caches clients. Ainsi, il ne peut exister plusieurs copies d'un même bloc, supprimant tout problème de cohérence.

Dans le système GMS [95], les pages injectées en mémoire globale sont en copie unique. La gestion de la cohérence des blocs entre les différents caches locaux est laissée à la charge des couches systèmes de niveau supérieur.

4.3.2.4 Jeton

Les systèmes de cohérence à **jeton** sont très proches des mécanismes de cohérence séquentielle que l'on rencontre dans les mémoires partagées réparties (voir chapitre 3.2.3). L'objectif est d'autoriser N accès simultanés en lecture **ou** un seul accès en écriture. Lorsqu'un nœud désire écrire dans un fichier, il doit tout d'abord obtenir le jeton d'écriture pour ce fichier. Lorsqu'un nœud dispose du jeton il peut écrire dans le fichier à loisir, sans demander aucune autre autorisation. Lorsqu'un autre nœud demande le jeton, le nœud propriétaire invalide sa copie du fichier et transmet le jeton au nouveau propriétaire.

Cette solution est très largement utilisée avec cependant des variantes concernant la granularité du maintien de la cohérence. AFS [68] et Coda [86] utilisent le fichier comme unité de cohérence. xFS [7] gère la cohérence au niveau des blocs du cache de fichiers.

4.4 Partage physique de la ressource disque

Le partage physique de la ressource disque est réalisé suivant deux approches totalement différentes par les systèmes de gestion de fichiers distribués et parallèles. Nous détaillons ces deux approches dans la suite de ce paragraphe.

4.4.1 Cas des SGFD

Dans un SGFD, le partage physique des disques est réalisé en autorisant un processus à stocker un fichier sur un disque distant. Dans le cas de NFS [84] par exemple, il est possible pour un processus s'exécutant sur un nœud donné de stocker des fichiers sur les disques de nœuds distants grâce au mécanisme de montage. Chaque disque distant est monté sur un répertoire différent à travers lequel il est possible d'accéder à ce disque. L'utilisation des disques distants est cependant réalisée explicitement par l'utilisateur en stockant un fichier dans un répertoire de montage plutôt que dans un répertoire correspondant à un disque local.

Dans AFS [68] ou xFS [7] un fichier peut être stocké sur n'importe quel disque de l'architecture sans demande explicite de l'utilisateur. De plus, les fichiers peuvent être automatiquement migrés du disque d'un nœud vers le disque d'un autre nœud, afin d'équilibrer la charge de stockage des disques ou de diminuer la consommation de bande passante réseau en rapprochant un fichier de son utilisateur.

4.4.2 Cas des SGFP

Dans un SGFP, le partage physique des disques est réalisé en répartissant le stockage des données d'un fichier sur un sous-ensemble ou sur la totalité des disques d'une architecture.

La distribution des données sur les disques est un problème important lors de la conception d'un SGFP car une mauvaise distribution peut conduire à des accès séquentiels aux fichiers. Une bonne distribution des données doit permettre une utilisation optimale des accès en parallèle. Il faut pour cela déterminer la taille des fragments à utiliser et la répartition de ceux-ci sur les disques. Les données peuvent être distribuées de plusieurs manières :

Étendue : Le fichier est découpé en d fragments, d étant le nombre de disques disponibles [69]. Chaque fragment est placé sur un disque différent.

Bloc : Le fichier est découpé en petits fragments de taille fixe. Ceux-ci peuvent être placés sur les disques de manière cyclique, c'est-à-dire que le fragment numéro n est placé sur le disque numéro n *modulo* d [65, 9]. D'autres méthodes de placement peuvent être utilisées via une interface spécialisée comme c'est le cas dans PPFS [50] ou MPI-IO [25]. Dans les systèmes xFS [7] ou Zebra [48] les fragments sont stockés automatiquement à la manière d'un disque RAID [77] sur un sous-ensemble des disques de l'architecture.

Structure 3-D : le système Galley [73] découpe un fichier en sous-fichiers distribués sur l'ensemble des disques et découpe chaque sous-fichier en **fork** correspondant à une séquence d'octets de taille variable. Le nombre de sous-fichiers est fixé lors de la création du fichier principal. La taille des sous-fichiers, ainsi que le nombre et la taille des forks sont libres et peuvent changer dynamiquement durant la vie du fichier.

4.5 Gestion globale des caches de fichiers

Même si le service de cache de fichiers n'est pas directement accessible aux utilisateurs, il peut être intéressant de lui apporter les propriétés de transparence énoncées dans le paragraphe 2.3.2. Ces propriétés s'expriment alors de la manière suivante :

Transparence de localisation Un système de fichiers peut accéder de la même manière à un bloc qu'il soit stocké dans le cache de fichiers local ou dans un cache distant.

Indépendance de localisation Le déplacement d'un bloc d'un cache de fichiers vers un autre est transparent pour le système de fichiers.

Partage logique Les données présentes dans le cache de fichiers d'un nœud peuvent être utilisées par tous les nœuds de la grappe.

Partage physique Les données d'un fichier peuvent être stockées dans le cache de fichiers d'un nœud distant.

Les propriétés de transparence et d'indépendance à la localisation ne sont pas nécessaires pour fournir un système à image unique puisque le service de cache de fichiers est enfoui dans un SGF, SGFD ou SGFP et donc invisible pour l'utilisateur. De plus, ces propriétés sont intimement liées à la mise en œuvre des systèmes de gestion de fichiers les intégrant. C'est pourquoi nous ne détaillons pas ces propriétés dans ce paragraphe.

En revanche, si les propriétés de partage logique et de partage physique ne sont pas non plus nécessaires pour offrir un SSI fondé sur une gestion globale des disques, elles

permettent une amélioration significative des performances d'un SGF distribué ou parallèle. La mise en œuvre de ces propriétés est connue sous le nom de système de caches de fichiers coopératifs [30].

Après avoir donné une définition d'un cache de fichiers coopératif, nous détaillons dans ce paragraphe les mécanismes de partage physique et logique d'un cache de fichiers et présentons les solutions proposées dans la littérature afin de résoudre les problèmes de cohérence des données dupliquées dans les caches.

4.5.1 Problématique et définition

Les SGFD et les SGFP utilisent intensivement les mécanismes de caches de fichiers afin de masquer les faibles performances des unités de stockage. Typiquement, on trouve un cache disque sur chaque nœud, dont la taille est limitée par la quantité de mémoire physique disponible sur le nœud. Ces caches sont indépendants et gérés localement sur chaque nœud. Or, sur une architecture distribuée il peut être intéressant de gérer les caches situés sur les différents nœuds comme un seul et unique cache de grande taille : un **cache coopératif** [30]. On peut ainsi espérer diminuer le nombre de défauts grâce à l'augmentation de la taille du cache et exploiter les blocs placés en cache par des processus s'exécutant sur des nœuds distants.

Définition 20 (Cache coopératif) *Un **cache coopératif** est un service système assurant le partage et la coordination des blocs présents dans les caches de fichiers d'un ensemble de nœuds (clients et/ou serveurs).*

4.5.2 Partage physique d'un cache de fichiers

Le partage physique d'un cache de fichiers consiste à utiliser la mémoire de nœuds distants comme une extension du cache local grâce à un mécanisme de cache coopératif privé (voir figure 4.5.a). Lorsqu'un nœud actif doit évincer des blocs de son cache, il les injecte dans la mémoire d'un nœud distant inactif. Le nœud actif peut alors accéder à des blocs situés dans ce cache privé distant ou dans son cache local. Lorsque le nœud inactif redevient actif, il vide de sa mémoire les blocs du cache coopératif. Cette solution est très proche des mécanismes de pagination à distance (voir chapitre 3.1.2). Le système de pagination à distance GMS dispose d'ailleurs d'un cache coopératif privé reposant sur l'injection de pages à distance [95].

En se limitant au partage physique d'un cache de fichier, un cache de fichiers coopératif souffre d'importantes limitations. Dans cette situation, les caches distants sont utilisés comme une extension du cache local des noeuds. Cependant, le SGF ne peut pas utiliser les blocs placés dans son cache local par un nœud distant, ce qui réduit le taux de succès potentiel.

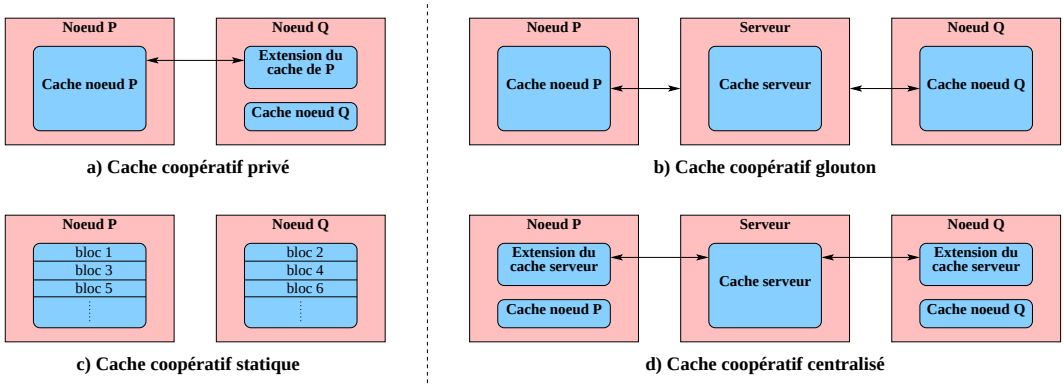

FIG. 4.5 – *Différentes mises en œuvre d'un cache coopératif*

4.5.3 Partage logique d'un cache de fichiers

La plupart des mécanismes de cache de fichiers coopératif mettent en œuvre à la fois la propriété de partage physique et la propriété de partage logique. Cependant, la gestion de ces caches est réalisée de manière très différente suivant les systèmes. Nous présentons dans ce paragraphe un aperçu des principales solutions présentées dans la littérature.

4.5.3.1 Cache coopératif glouton

Au sein d'un cache coopératif glouton, chaque nœud place des blocs dans son cache de manière classique. Aucune coordination du contenu des caches locaux n'est assurée. Lorsqu'un défaut de cache intervient sur un nœud P, une requête est envoyée au serveur de fichier. Si le serveur de fichiers dispose du bloc dans son cache, il est retourné au nœud P. Sinon, le serveur consulte une structure de données indiquant le contenu des caches de chaque client. Si un client dispose du bloc, la requête lui est transmise, sinon le bloc est lu depuis le disque et envoyée au nœud P. Le serveur mémorise alors que ce bloc est présent dans le cache local du nœud P.

Cet algorithme permet de partager le contenu des caches de l'ensemble des nœuds du système avec une modification très légère du système. Cependant, le manque de coordination des caches locaux peut conduire à des duplications inutiles de blocs, gaspillant la ressource mémoire. De plus, le serveur centralisé représente un goulot d'étranglement important, préjudiciable en terme de performance.

4.5.3.2 Cache coopératif statique

Dans le système PAFS, Cortes et al. proposent un système de caches coopératifs, assurant une gestion globale du contenu des caches locaux [29]. Il s'agit d'un cache associatif par ensemble : il y a autant d'ensembles que de nœuds et chaque ensemble est de la taille du cache local. Lorsqu'un bloc est placé dans le cache, une fonction de hachage détermine dans quel ensemble il doit être placé. Lors de la recherche d'un bloc dans le cache, la même

fonction de hachage est utilisée pour connaître son emplacement. Ainsi, à chaque bloc est associé un emplacement unique dans le système.

Cette politique résout le problème du gaspillage mémoire de la coopération gloutonne, supprime le serveur centralisé, offre une solution simple de localisation des blocs et supprime tout problème de cohérence en supprimant les copies multiples. Cependant, chaque accès à un bloc donné nécessite de contacter le nœud hébergeant le bloc, ce qui augmente le trafic réseau et exploite mal les principes de localité temporelle et spatiale.

4.5.3.3 Cache coopératif centralisé

Un cache coopératif centralisé est une approche intermédiaire entre le cache coopératif glouton et le cache coopératif privé. Le cache de chaque client est divisé en deux parties : le cache local géré localement par le nœud, et le cache global, utilisé comme extension du cache du serveur de fichiers. Lorsqu'un client P ne trouve pas un bloc dans son cache local, il envoie une requête au serveur. Le serveur vérifie si le bloc est présent dans son cache local. Si c'est le cas, il envoie le bloc au nœud P, sinon il vérifie si le bloc est présent dans une extension de son cache sur d'autres nœuds. Si un nœud distant héberge le bloc demandé, la requête lui est transmise, sinon le bloc est lu depuis le disque, envoyé au nœud P et placé dans le cache local du serveur. Lorsque le cache local du serveur est saturé, le bloc le plus ancien au sens LRU est injecté sur un nœud distant dans le cache global.

Cette solution permet d'assurer un taux de succès dans le cache global très élevé, grâce à une utilisation astucieuse de l'ensemble des mémoires du système. Cependant, la taille restreinte des caches locaux diminue sensiblement le taux de succès local. De plus, la coordination centralisée du cache provoque une charge importante du serveur.

4.5.3.4 Cache coopératif distribué

Dahlin et al. ont proposé un système de caches coopératifs distribué baptisé **N-Chance Forwarding** [30]. Cet algorithme modifie le mécanisme de coopération gloutonne en plaçant de préférence en cache des **singletons**. Un singleton est un bloc présent dans le cache d'un seul client de la grappe. Cet algorithme a pour but de limiter les copies d'un bloc tout en évitant l'éviction de singletons des caches des clients. Lorsqu'un client décide d'évincer un bloc, il vérifie s'il s'agit de la dernière copie du bloc. Si le bloc est un singleton, au lieu de le détruire, le client lui associe un **compteur d'injection** initialisé à N et l'injecte aléatoirement dans le cache d'un nœud distant. Si ce bloc est à nouveau évincé, son compteur d'injection est décrémenté et le bloc est injecté dans le cache d'un autre nœud. Lorsque ce compteur arrive à zéro, le bloc est détruit.

La valeur de N est un paramètre du cache. $N = 0$ correspond à un cache coopératif glouton. Cet algorithme permet d'éviter le gaspillage de mémoire tout en tirant partie de la localité temporelle. Si tous les nœuds accèdent régulièrement au même bloc, celui-ci est présent dans le cache local de chaque nœud. Si ce bloc n'est plus utilisé, il finit par être évincé de la mémoire de tous les nœuds au profit de nouveaux blocs. Néanmoins, la dernière copie n'est détruite qu'après avoir été évincée N fois, augmentant les chances

d'une réutilisation future.

4.6 Résumé

Nous avons vu dans ce chapitre qu'il existe deux approches permettant d'offrir les propriétés de transparence dans le cadre de la gestion de la ressource disque : les systèmes de gestion de fichiers distribués et les systèmes de gestion de fichiers parallèles. Le tableau 4.2 présente un résumé des propriétés offertes par différents systèmes mettant en œuvre ces mécanismes.

Un système de gestion de fichiers distribué permet à un processus d'accéder de manière transparente aux fichiers situés aussi bien sur son nœud d'exécution que sur un noeud distant, ceci grâce à l'interface standard d'accès aux fichiers. Suivant les systèmes, différentes approches sont utilisées pour désigner un fichier et le localiser dans une architecture distribuée. De cette désignation dépend la transparence et l'indépendance à la distribution. Les systèmes les plus aboutis, tel xFS, offrent une transparence et une indépendance totale à la localisation, respectant ainsi les propriétés nécessaires à la création d'un SSI. Néanmoins, les SGFD se révèlent peu efficaces lorsqu'ils sont utilisés dans un cadre de programmation parallèle, puisqu'ils limitent la bande passante d'accès aux fichiers à la bande passante d'un seul disque.

Un système de gestion de fichiers parallèle vise à offrir l'illusion d'un disque unique de grande capacité et à haut débit au dessus d'un ensemble de disques distribués, ce qui correspond exactement aux objectifs d'un SSI. Au sein d'un SGFP, un fichier est découpé en fragments stockés sur un ensemble de disques distribués. Les fragments peuvent ainsi être accédés en parallèle afin d'offrir un débit total élevé. Cependant, un SGFP est principalement destiné à la manipulation de fichiers de grande taille et se révèle peu performant lorsqu'il s'agit d'accéder à des fichiers de petite taille.

Enfin, ces deux catégories de SGF (SGFD et SGFP) utilisent des caches distribués afin de masquer la latence élevée et la faible bande passante des disques. Les mécanismes de caches coopératifs sont les versions les plus élaborés de ces caches distribués. Ils permettent d'utiliser la mémoire de l'ensemble des nœuds d'une architecture distribuée comme un seul cache de fichiers de très grande taille. Cette architecture de cache correspond très bien à la philosophie des systèmes à image unique. La réplication des données dans ces caches distribués pose cependant des problèmes de localisation et de cohérence. Les solutions les plus efficaces à ces problèmes sont proches des mécanismes utilisés dans les mémoires partagées réparties logicielles, à savoir une gestion de la cohérence par invalidation sur écriture des blocs du cache de fichier.

En résumé, les SGFD et SGFP remplissent les critères d'un SSI pour la gestion des disques distribués. Cependant, l'utilisation conjointe des mécanismes mis en œuvre par les deux types de système de gestion de fichiers est nécessaire si l'on désire offrir des accès performants aussi bien aux fichiers très volumineux qu'aux fichiers de petite taille. C'est ce que réalise le système xFS. Enfin, la principale critique que l'on peut faire à ces systèmes est qu'ils se focalisent sur la gestion des disques et n'offrent pas de système à image unique

global, c'est-à-dire offrant la vision d'une machine unique et pas uniquement la vision d'un disque unique.

	NFS	Andrew	Sprite	xFS
Transparence à l'utilisatoin	Oui	Oui	Oui	Oui
Transparence de localisation	Incomplet	Oui[2]	Oui	Oui
Indépendance de localisation	Non	Oui[3]	Oui	Oui
Partage physique	Incomplet[1]	Oui	Oui	Oui
Partage logique	Oui	Oui	Oui	Oui
Niveau de mise en œuvre	Noyau/Utilisateur	Noyau	Noyau	Noyau/Utilisateur
Sémantique de partage	Session	Session	Unix	Unix
Cohérence de cache	Vérification sur accès	Jeton	Pas de cache en écriture	Jeton
Granularité de cohérence	Fichier	Fichier	Fichier	Bloc

[1] Partage géré explicitement par l'administrateur via les points de montage.
[2] Le chemin d'accès à un fichier est différent suivant que le fichier est local ou global.
[3] Uniquement pour les fichiers globaux.

TAB. 4.2 – Comparaison des propriétés offertes par différents systèmes de gestion globale des disques

5 GESTION GLOBALE DE LA RESSOURCE PROCESSEUR

Si l'on considère un SSI fondé sur une gestion globale de la ressource processeur, les propriétés de transparence définies au paragraphe 2.3.2 s'expriment de la manière suivante :

Transparence de localisation : un processus s'exécute de la même manière quelque soit le nœud sur lequel il a été lancé.

Indépendance de localisation : le déplacement d'un processus d'un nœud d'exécution vers un autre est transparent pour l'application.

Partage logique : les *threads* d'un processus peuvent s'exécuter sur des nœuds distincts.

Partage physique : les processus créés ou s'exécutant sur un nœud d'une grappe peuvent être exécutés sur le processeur de nœuds distants.

Tout système distribué composé de nœuds d'architecture équivalente et disposant d'un environnement logiciel homogène offre la propriété de transparence de localisation. Cette propriété est offerte depuis plusieurs décennies par tous les systèmes d'exploitation, c'est pourquoi nous ne jugeons pas utile de la décrire dans ce chapitre.

La propriété de partage physique peut être réalisée grâce à plusieurs mécanismes. Cependant, nous nous intéressons principalement dans ce chapitre aux mécanismes de placement et de migration de processus. La migration de processus consiste à déplacer un processus d'un nœud vers un autre au cours de son exécution.

La propriété d'indépendance de localisation n'a de sens que dans le cadre de la migration de processus. Cette propriété est assurée avec plus ou moins de succès par tous les systèmes offrant un service de migration de processus. C'est de cette propriété que dépend réellement la transparence à la migration.

Nous présentons dans ce chapitre les mécanismes proposés afin de réaliser les propriétés de partage physique de la ressource processeur, d'indépendance à la localisation et de partage logique de processus. Le lecteur non familier avec les mécanismes système de gestion des processeurs peut se reporter à l'annexe A.3 qui présente un rappel des principes de fonctionnement d'un système d'exploitation.

5.1 Partage physique de la ressource processeur

La propriété de partage physique est apparue très tôt dans les systèmes de type Unix^{TM}, qui offrent la possibilité de se connecter a des nœuds distants afin d'y lancer l'exécution

de processus (suite à une commande de type « *rlogin* » ou encore « *rsh* »). Néanmoins, les mécanismes d'exécution à distance fournis par Unix^{TM}n'offrent aucune transparence puisque l'utilisateur doit explicitement demander l'exécution distante et préciser sur quel nœud cette exécution doit avoir lieu.

Des mécanismes plus évolués ont vu le jour afin d'automatiser l'exécution de processus à distance. Les systèmes de **batch** tels que PBS [49] par exemple, permettent d'exécuter automatiquement un ensemble de processus sur un parc de machines. Ces outils fonctionnent sur un modèle de files d'attente avec priorités. Les priorités d'exécution des processus sont fixées explicitement par les utilisateurs, généralement suivant des critères politiques et économiques.

Dans le domaine de la programmation parallèle, des systèmes tels que PVM [97] offrent des mécanismes de création de processus à distance. Les processus créés sont automatiquement placés sur les nœuds disponibles de l'architecture parallèle suivant des heuristiques internes au système ou définies par l'utilisateur.

Cependant, ces solutions ne répondent pas aux critères de transparence que nous souhaitons atteindre. D'autres solutions permettant la création, le placement et éventuellement le déplacement (migration) automatique de processus ont été proposées. Nous détaillons ces solutions de gestion et d'ordonnancement global des processus dans la suite de ce paragraphe.

5.1.1 Ordonnancement de processus sur une grappe

Dans le contexte d'une grappe de calculateurs, chaque nœud dispose d'un système d'exploitation assurant l'ordonnancement des processus de ce nœud sur le processeur local. Celui-ci s'assure de ne pas laisser le processeur local inactif et d'exécuter les processus présents dans sa file d'attente. Or, si l'ordonnancement est optimal localement, il ne l'est pas à l'échelle de la grappe. En effet, un nœud A peut héberger un grand nombre de processus actifs alors qu'un nœud B est inactif par manque de processus à exécuter. Localement, l'ordonnancement est efficace, mais il est évident qu'il pourrait être globalement plus efficace en permettant l'exécution de certains processus du nœud A sur le processeur du nœud B, ne laissant ainsi aucun processeur inactif.

Définition 21 (Placement) *On désigne par **placement**, le choix initial d'un nœud d'exécution pour un processus nouvellement créé.*

La gestion de processus sur une grappe de calculateurs nécessite de choisir à la fois les nœuds où les processus vont être exécutées (**placement**) et leur allocation temporelle (**ordonnancement**). Historiquement, le placement intelligent des processus à exécuter a été la première approche pour la gestion globale des processus. Sur les premiers systèmes distribués équipés d'un réseau à faible débit, **migrer** un processus, c'est-à-dire changer son nœud d'exécution, était d'un coût trop important. On s'est donc attaché à placer efficacement les processus sur les nœuds lors de leur création. Cette solution est cependant loin d'être satisfaisante, car la charge globale des processeurs peut ne pas être équilibrée une fois les processus démarrés sur leur nœud d'exécution respectif.

L'augmentation des performances des réseaux locaux, en terme de latence et de bande passante et l'utilisation d'algorithmes plus performants a permis de diminuer de manière significative le coût de la migration d'un processus, et ainsi de pouvoir raisonnablement envisager son utilisation dans le cadre de l'ordonnancement sur grappe.

Définition 22 (Migration) *On appelle **migration**, l'opération consistant à déplacer un processus du nœud ou il est en cours d'exécution sur un autre nœud.*

Grâce à la migration de processus, il est possible de changer le nœud d'exécution d'un processus durant son exécution et ainsi d'équilibrer dynamiquement la charge globale des processeurs d'une grappe. Dans un système distribué, il est également important de tenir compte de l'occupation mémoire des processus afin d'éviter le remplacement de pages sur disque, opération coûteuse et source d'écroulement des performances[2].

5.1.1.1 Répartition de la charge

Pour utiliser au mieux un système distribué, il faut répartir la charge sur les différents nœuds du système et éviter l'inactivité des processeurs. Deux approches sont possibles : l'**équilibrage de charge** et le **partage de charge**. Dans l'équilibrage de charge, le but est de donner une charge uniforme aux nœuds du système, il s'agit de minimiser l'écart à la moyenne sur tous les nœuds [67]. Dans le partage de charge, on s'assure simplement qu'aucune machine n'est sous- ou surchargée. Par exemple, Condor [4] et Mosix [3] gèrent le partage de charge en permettant de migrer un processus sur une machine inoccupée.

5.1.1.2 Utilisation maximale de la mémoire

Un autre objectif important, lorsque le réseau est plus rapide qu'un disque, est d'éviter de devoir stocker des pages sur disque à cause d'un manque de mémoire. Mosix [2] teste régulièrement la quantité de mémoire disponible sur chaque nœud. Lorsque celle-ci est en dessous d'un certain seuil, un processus est migré sur une machine qui dispose encore de suffisamment de mémoire.

On peut remarquer que cette politique n'est pas en contradiction avec l'équilibrage de charge. On peut par exemple privilégier l'équilibrage de charge processeur tant que l'utilisation mémoire n'est pas critique et ne tenir compte de la charge mémoire que lorsqu'un nœud devient sous- ou surchargé [2].

5.1.2 Éléments de mise en œuvre

La migration de processus est un mécanisme complexe à mettre en œuvre. Un processus est composé de nombreux éléments qu'il est nécessaire de prendre en compte durant la migration [33] :

- l'**état du processus** constitué des données sauvegardées par le noyau lors d'un changement de contexte ;

- les **données noyau liées au processus** tel que l'identifiant de processus, les références aux processus parents et fils, le répertoire courant, les masques de signaux, etc ;
- l'**espace d'adressage virtuel**, c'est-à-dire l'ensemble des données projetées en mémoire virtuelle ;
- la **liste des fichiers ouverts**. Les données de ces fichiers se trouvent dans l'espace d'adressage du processus, mais également dans les caches et tampons du noyau. De plus, l'état des fichiers en cours d'utilisation est stocké dans les structures noyaux des processus les utilisant et dans les structures noyaux d'interface avec le matériel ;
- l'**état des canaux de communication**. Les données des canaux se trouvent dans l'espace d'adressage du processus et dans les tampons du système. L'état des canaux de communication est stocké dans les structures des processus les utilisant et dans les structures noyau d'interface avec le matériel ;
- l'**accès aux autres périphériques** tel que le clavier, la souris, la carte vidéo, l'imprimante, etc.

On peut décomposer la migration d'un processus en deux parties distinctes : (1) le déplacement de son contexte d'exécution et (2) la liaison du processus migré avec les ressources qu'il utilisait sur son nœud d'exécution initial. La première partie représente la partie visible de la migration, c'est-à-dire le déplacement d'une activité d'un nœud vers un autre. Les éléments à prendre en compte pour cette phase sont l'état du processus et les données du noyau liées à ce processus. Néanmoins, la migration du contexte d'exécution ne suffit pas à permettre le redémarrage d'un processus sur un nouveau nœud. Lorsqu'un processus s'exécute sur un nœud, il dispose d'un environnement d'exécution lui permettant d'avoir accès à la mémoire, aux fichiers et aux différents périphériques de ce nœud. Lorsqu'un processus est migré, il est nécessaire de lui offrir le même environnement d'exécution afin qu'il puisse continuer son exécution et que la migration ne soit pas perceptible par l'utilisateur. Maintenir un environnement d'exécution homogène pour un processus après sa migration représente la seconde phase de la migration. Les éléments à prendre en compte pour cette phase sont : l'espace d'adressage du processus, la liste des fichiers ouverts, l'état des canaux de communication et l'accès aux périphériques. Les mécanismes mis en jeu dans cette seconde phase assurent l'indépendance à la localisation. Nous présentons ces mécanismes dans la suite de ce chapitre.

5.2 Indépendance à la localisation

Un processus utilise de nombreuses ressources logiques et physiques, créant un lien avec son nœud d'exécution. Lors de la migration d'un processus, il faut s'assurer qu'il puisse toujours disposer des ressources qu'il utilisait sur son nœud initial. La diversité et le nombre de ressources pouvant être utilisées par un processus implique de nombreux traitements lors de la migration et éventuellement après la migration. Nous présentons dans la suite de cette partie un aperçu des solutions permettant d'assurer l'indépendance de localisation nécessaire à la migration de processus.

5.2.1 Problèmes de l'accès aux ressources

Le problème majeur lors de la migration de processus est de maintenir le lien entre le processus migré et les composants logiques ou physiques qu'il utilisait sur son nœud d'exécution initial. Pour chaque composant, trois alternatives sont possibles : (1) migrer le composant sur la machine cible (voir figure 5.1.a), (2) rediriger les requêtes du processus migré vers le composant laissé sur la machine source (voir figure 5.1.b) ou (3) supprimer la liaison avec le composant, au détriment de la transparence.

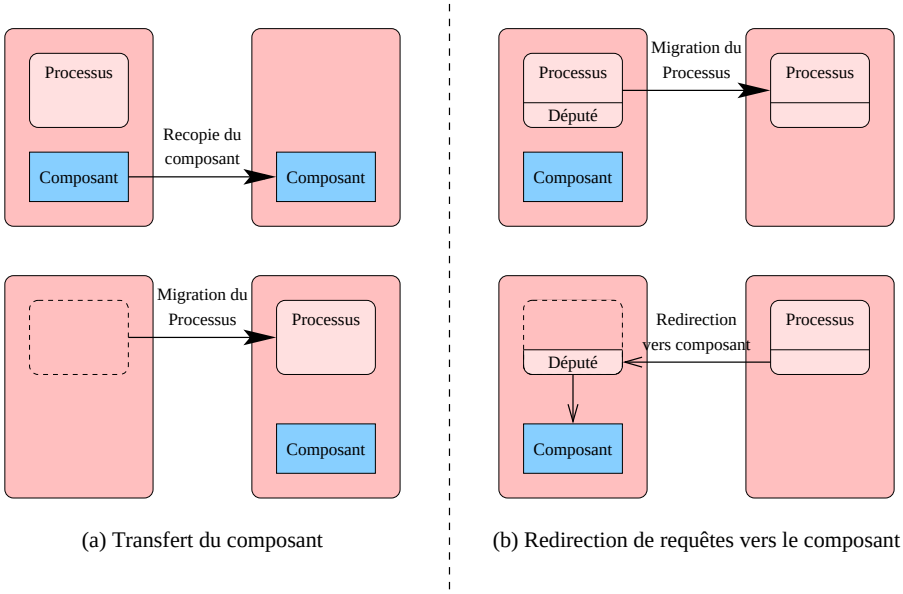

FIG. 5.1 – *Migration des composants d'un processus*

5.2.1.1 Migration de composant

Pour migrer un composant, il faut l'extraire de la machine source, le transférer vers la machine cible et le réimplanter dans le noyau en reconstruisant les liens avec le processus migré et les données connexes du noyau. Par exemple, il est relativement simple de transférer l'état du processus ou l'espace d'adressage puisqu'il ne s'agit que de données ayant peu ou pas d'interaction avec les autres processus et stockées uniquement en mémoire. Il est en revanche beaucoup plus complexe de transférer la liste des fichiers ouverts, qui dépendent de données stockées sur les disques de la machine source et pouvant être partagées avec d'autres processus.

5.2.1.2 Redirection de requêtes

La redirection des requêtes d'accès aux composants se déroule de la manière suivante :
– Détection de l'accès au composant par le processus migré ;

- Transfert de la requête d'accès à la machine source ;
- Accès réel au composant sur la machine source ;
- Transfert du résultat au processus migré de la machine cible.

Cette solution est souvent retenue pour l'accès aux périphériques de la machine source, ainsi que pour certains appels systèmes tel que la communication avec les processus père et fils, l'obtention de la date système, etc... Elle implique cependant une transaction réseau lors de chaque accès à composant situé sur la machine source et donc un surcoût pouvant être pénalisant.

5.2.1.3 Suppression de liaison

Enfin, la dernière solution consiste à supprimer la liaison entre le processus et le composant source lorsqu'aucune des deux solutions précédentes n'est envisageable. C'est notamment le cas pour l'accès au *frame buffer* d'une carte vidéo. Dans ce cas, le processus migré pourra soit accéder au *frame buffer* de la machine cible soit ignorer tout accès à celui-ci, altérant ainsi la propriété de transparence à la migration. En dernier recours, la migration peut être prohibée si l'accès à la ressource ne peut être supprimé et que la propriété de transparence doit impérativement être conservée. On comprend aisément qu'un processus assurant un affichage graphique de haute qualité ne peut être migré. Écrire dans le *frame buffer* de la machine cible supprimerait l'affichage sur la machine source et relayer les données du processus migré vers le *frame buffer* source serait d'un coût réseau déraisonnable.

5.2.2 Migration de l'espace d'adressage

La migration de l'espace d'adressage d'un processus est sans aucun doute le sujet le plus abordé dans la littérature. Tout d'abord parce tout processus, même très simple, a besoin d'un espace d'adressage. Ensuite parce que le temps de transfert de l'espace d'adressage peut être un frein à la migration.

5.2.2.1 Recopie de l'image mémoire

La première solution au problème de transfert de l'espace d'adressage virtuel a été de recopier intégralement l'image mémoire d'un processus de la machine source vers la machine cible, comme c'est le cas dans Condor [4] ou Charlotte [10] (voir figure 5.2.a). Cette approche, bien que simple, souffre de deux inconvénients majeurs. Tout d'abord, le temps de transfert de l'image mémoire du processus peut être élevé, de l'ordre de quelques secondes, durant lesquelles le processus ne peut plus s'exécuter ni sur la machine source, ni sur la machine cible. Le second problème est le gaspillage de travail et de bande passante réseau lorsqu'une grande partie des données transférées ne sont pas utilisées par le processus migré. Afin de limiter le surcoût occasionné par le transfert de l'espace d'adressage, plusieurs solutions ont été apportées.

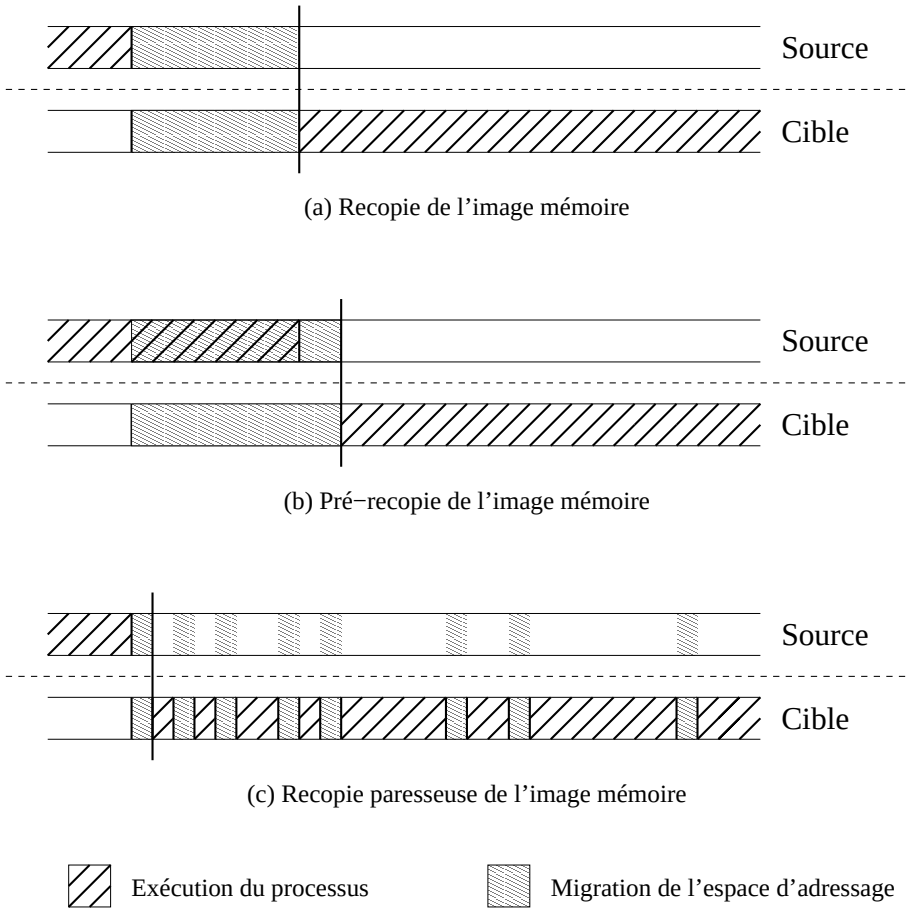

FIG. 5.2 – *Techniques de migration de l'espace d'adressage d'un processus*

5.2.2.2 Pré-copie de l'image mémoire

Le système V [21] résout ce problème en effectuant une *pré-copie* de l'image mémoire du processus (voir figure 5.2.b). Plutôt que de bloquer un processus durant la transmission de son image mémoire, le système V permet au processus de continuer son exécution sur le nœud source durant la recopie de son image mémoire. Les données modifiées par le processus durant la copie sont transférées à nouveau sur la machine cible juste avant la migration du processus. Cette méthode permet de limiter le laps de temps durant lequel un processus à migrer ne peut s'exécuter. Il est ici réduit au transfert des pages modifiées durant la pré-copie. Cependant, subsiste le problème des copies inutiles : celles effectuées pour les pages qui ne seront jamais utilisées sur la machine cible et celles copiées deux fois pour cause de modification durant la pré-copie.

5.2.2.3 Recopie paresseuse de l'image mémoire

Accent [36] ou Mosix [3] utilisent une approche de copie paresseuse afin de résoudre les problèmes de copie inutile (voir figure 5.2.c). Avec cette approche, un processus est migré et redémarré sur le nœud cible sans qu'aucune donnée de son espace d'adressage ne soit transférée. Les données sont transférées à la demande lorsqu'elle ne sont pas présentes sur le nœud cible, grâce au mécanisme de pagination du système d'exploitation. Ainsi, les données qui ne sont pas utilisées par le processus après migration ne sont jamais transférées. De plus le temps d'inactivité du processus lors de la migration est minimum. Mais le principal problème de cette solution est le surcoût logiciel nécessaire à la migration paresseuse des pages et aux résidus du processus sur la machine source, résidus pouvant subsister sur plusieurs machines dans le cas de migrations successives d'un même processus. Ce phénomène complexifie encore la mise en œuvre du système.

Sprite [33] utilise une autre solution qui consiste à sauvegarder l'image du processus dans un fichier partagé avant la migration. Il est ainsi possible de faire disparaître toute trace du processus sur la machine source, tout en permettant un chargement des pages à la demande simple sur la machine cible. Le chargement est simple puisque reposant sur les mécanismes standard du système d'exploitation. Cependant, bien que Sprite dispose d'un système de fichiers distribué très efficace, reposant sur l'utilisation intensive de caches [98], la sauvegarde des données modifiées d'un processus sur un serveur de fichiers reste très coûteuse.

5.2.3 Migration des fichiers ouverts

La migration des fichiers ouverts pose de nombreux problèmes d'ordre technique. Comme nous l'avons vu au chapitre 4, la mise en œuvre d'un système de fichier centralisé ou distribué est complexe. De nombreuses structures de données et zones de stockage intermédiaires (tampons, caches, etc) sont nécessaires à son fonctionnement. Dans le cadre de la migration de processus, il est nécessaire de distinguer deux types d'accès aux fichiers : l'accès aux **fichiers locaux** par l'intermédiaire d'un SGF centralisé et l'accès aux **fichiers distants** par l'intermédiaire d'un SGFD. On peut alors mettre en évidence les problèmes suivants :

Accès distant à un fichier local Lorsqu'un processus est migré de son nœud d'origine A vers le nœud cible B, le fichier F qui est local sur A et accédé via un SGF centralisé devient distant sur B. L'accès à ce fichier doit être maintenu.

Référence interne au fichier Au sein du système d'exploitation, l'identification d'un fichier est réalisée grâce à un nom interne local. Lors de la migration d'un processus, cette désignation doit rester cohérente et compréhensible par le système sur le nœud cible.

Pointeur de fichier Soit deux processus A et B partageant le même pointeur de fichier suite à un *fork* ou à la création d'un *thread*. Après la migration de B, le pointeur est toujours partagé par A et B, mais les deux processus s'exécutent sur des nœuds différents. Où placer ce pointeur et comment assurer une gestion distribuée de celui-ci ?

Cache de fichiers Lorsqu'un processus accède à un fichier, les données référencées sont placées dans le cache de fichiers, qui peut contenir des données modifiées par le processus avant sa migration. Ces modifications doivent rester visibles et cohérentes après la migration du processus.

Serveur avec état Lorsqu'un processus accède à des fichiers distants via un SGFD, le serveur de fichiers peut détenir des informations sur ce processus (nœud d'exécution, liste des blocs accédés, etc). La migration du processus peut introduire des incohérences au sein des informations manipulées par le serveur de fichier.

Les composants à prendre en compte lors de la migration d'un fichier local sont :
- les structures de données noyau liées au fichier (nom interne, pointeur de fichier, etc) ;
- l'état du cache de fichier ;
- le fichier sur disque.

En pratique, le fichier sur disque n'est jamais migré. Le coût d'une telle opération est en effet très élevé, notamment à cause du faible débit des disques.

Les composants à prendre en compte lors de la migration d'un fichier distant sont :
- les structures de données noyau liées au fichier (référence du serveur de fichier, nom interne, pointeur de fichier, etc) ;
- l'état du cache de fichier local ;
- les structures de données du serveur.

5.2.3.1 Redirection des accès aux fichiers

Le système Mosix [3] ne migre aucun composant et utilise exclusivement un mécanisme de redirection des requêtes. Tous les appels système relatifs aux fichiers sont détournés et transmis à un **député** situé sur la machine source. Celui-ci transmet la requête au système d'exploitation comme si elle provenait d'un processus local. Le résultat de l'opération est retourné au processus migré par l'intermédiaire du député. Cette méthode très simple résout tous les problèmes évoqués précédemment puisque le système d'exploitation de la machine source a l'illusion que le processus effectuant les opérations d'entrées/sorties n'a pas quitté le nœud et continue à s'exécute localement. Cependant, cette méthode souffre de nombreux problèmes de performance. Tout d'abord, chaque accès au fichier nécessite une transaction sur le réseau pour atteindre le site source. Ensuite, l'accès à un fichier situé sur un serveur nécessite également de passer par le site source, alors que la requête pourrait être transmise au serveur directement depuis le site d'exécution. Enfin, le processus migré ne dispose plus de cache de fichiers local puisque tous les accès aux fichiers sont systématiquement détournés vers le nœud source.

5.2.3.2 Utilisation d'un système de gestion de fichiers global

La migration des fichiers ouverts dans le système Sprite [33] repose sur un SGFD faisant partie intégrante du système (voir chapitre 4). Tous les accès aux fichiers passent par ce système de fichiers qui assure une grande transparence et indépendance vis-à-vis

de la localisation des fichiers et des processus y accédant. Nous détaillons brièvement les solutions retenues dans le système Sprite pour assurer la migration des fichiers ouverts.

Accès distant à un fichier local Les machines composant un système Sprite ne disposent pas de disque local. Tous les accès aux fichiers sont donc distants.

Référence interne au fichier Les fichiers ouverts lors de la migration sont fermés sur le site source et ré-ouverts sur le site cible, produisant ainsi de nouvelles références internes, ayant un sens sur ce nœud cible.

Pointeur de fichier Lorsqu'un fichier est partagé par plusieurs processus, le pointeur de fichier est géré par le serveur de fichier, centralisant ainsi la gestion de celui-ci.

Cache de fichiers Les données du cache modifiées par un processus sont transmises au serveur de fichier avant la migration du processus. Le serveur dispose alors des dernières modifications apportées au fichier. Celles-ci pourront être transmises au processus migré suite à des défauts de cache sur le nœud cible.

Serveur avec état Les états internes du serveur de fichiers sont mis à jour par le mécanisme de migration grâce à une primitive spécialisée. Ceci est possible grâce à la mise en œuvre conjointe au sein de Sprite du SGFD et du mécanisme de migration.

5.3 Partage logique de processus

Le partage logique de processus consiste à pouvoir exécuter les *threads* d'une application sur différents processeurs d'une architecture. Cette fonctionnalité est présente dans tous les systèmes d'exploitation s'exécutant sur des architectures multiprocesseur à mémoire partagée depuis de nombreuses années. Cependant, la réalisation de ce mécanisme sur une architecture à mémoire distribuée est relativement récente. Au sein d'une application, les *threads* partagent le même environnement d'exécution, notamment l'espace d'adressage virtuel et l'ensemble des fichiers ouverts. Cet environnement offert par les systèmes d'exploitation pour machines SMP n'est pas présent sur les architectures distribuées de type grappe. Pour permettre l'exécution distribuée de *threads* sur une grappe, il est donc nécessaire de concevoir un environnement d'exécution incluant le partage de mémoire et de fichiers à travers les nœuds d'une grappe.

Des systèmes tels que Millipede [52], D-CVM [91] ou encore DSM-Threads [70] utilisent une MPRL comme mécanisme de base permettant de partager la mémoire d'une grappe et ainsi d'exécuter des *threads* à travers les nœuds de cette grappe. Dans ces systèmes, l'espace d'adressage d'un processus est composé de deux types de données : (1) les données non partagées (la pile, le tas) et (2) les données partagées via un segment de mémoire gérée par la MPRL. Les segments de mémoire partagée utilisés par les processus sont visibles et accessibles depuis tous les nœuds du système. Lorsqu'un processus est migré, le système recopie l'intégralité des données non partagées, mais ne recopie aucune donnée partagée. Les données partagées sont recopiées automatiquement par la MPRL (en tenant compte du protocole de cohérence) lorsque le processus migré y accède.

5.4 Résumé

Nous avons présenté dans ce chapitre les mécanismes permettant d'assurer les propriétés de transparence dans le cadre de la gestion de processeurs en nous focalisant sur les mécanismes de migration de processus. Le tableau 5.1 présente un résumé des propriétés offertes par différents systèmes mettant en œuvre ces mécanismes.

De nombreux systèmes ont mis en œuvre des mécanismes de migration de processus avec pour objectif de rendre ce mécanisme le plus efficace et le plus transparent possible à l'utilisateur. Pour atteindre ces objectifs, des mécanismes logiciels complexes doivent être mis en œuvre afin de permettre une migration paresseuse et efficace de l'espace d'adressage, de maintenir le lien avec les fichiers ouverts et les périphériques de la machine source, etc. Ceci implique une modification lourde du système d'exploitation sous-jacent limitant la portabilité et la maintenabilité du système.

De plus, même dans un système très abouti, tel Mosix, la vision d'une machine unique est très imparfaite. Tout d'abord, l'absence de mécanisme de mémoire partagée interdit la migration de *threads* à travers la grappe et limite la programmation d'applications parallèles à un modèle par échange de messages. Les processus échangeant des données via des segments de mémoire partagée (de type system V) ou les *threads* d'un processus doivent obligatoirement être localisés sur le même site.

L'accès aux fichiers distants et fichiers ouverts avant la migration est également un problème difficile et dont les réalisations sont à ce jour non satisfaisantes en terme de performance et de transparence. Ces problèmes sont principalement liés à la mise en œuvre d'un mécanisme de gestion globale des processeurs dans un environnement où les autres ressources sont gérées localement.

	Niveau de mise en œuvre	Transparence à l'utilisation	Transparence de localisation	Indépendance de localisation	Partage physique	Partage logique
Condor	Utilisateur	Non[a]	Oui	Incomplet	Oui	Non
Sprite	Noyau	Oui	Oui	Oui	Oui	Non
Mosix	Noyau	Oui	Oui	Oui	Oui	Non
Millipede	Utilisateur	Non[ab]	Oui	Incomplet[1]	Non	Oui
D-CVM	Utilisateur	Non[a]	Oui	Incomplet	Non	Oui

[a] L'utilisation de ce système nécessite l'insertion de primitives spécifiques dans les applications.
[b] L'utilisation de ce système impose de nombreuses restrictions d'utilisation.
[1] Pas d'appels systèmes, pas d'utilisation de bibliothèques dynamiques, placement explicite des données partagée en MPRL.

TAB. 5.1 – Comparaison des propriétés offertes par différents systèmes de gestion globale de la ressource processeur

6 SYNTHÈSE

Les grappes de calculateurs sont constituées d'un ensemble de nœuds interconnectés par un réseau. Au sein de cette architecture, toutes les ressources sont distribuées et aucun mécanisme matériel ou logiciel ne permet d'utiliser simplement l'ensemble de ces ressources. Pour tirer parti de la puissance potentielle d'une grappe, le programmeur doit gérer manuellement l'utilisation des ressources distribuées, ce qui nécessite l'utilisation de méthodes de programmation complexes que seuls des spécialistes sont capables d'appréhender correctement. Cette complexité d'utilisation est un frein à l'expansion de l'utilisation des grappes.

Pour simplifier la programmation des grappes, de nombreux mécanismes fondés sur une gestion globale des ressources ont été proposés. Ces mécanismes permettent de donner l'illusion d'une ressource unique et partagée au dessus d'un ensemble de ressources distribuées. Cependant, la plupart des systèmes existants se focalisent sur la gestion d'un seul type de ressource (processeur, mémoire ou disque) ce qui ne permet pas d'offrir un système à image unique satisfaisant. De plus, dans ce genre de systèmes, de nombreuses contraintes apparaissent à la frontière entre une ressource gérée globalement et une ressource gérée localement. Par exemple dans le système Mosix [3], des contraintes apparaissent à la frontière entre la gestion globale des processeurs et la gestion locale de la mémoire et des disques. Lorsqu'un processus est migré, l'absence de système de gestion globale des disques introduit des problèmes de performance lors de l'accès à des fichiers distants, en interdisant notamment l'utilisation du cache de fichiers local. D'autre part, l'absence de mécanisme de gestion globale de la mémoire interdit l'exécution des *threads* d'une application sur différents nœuds d'une grappe.

Une grappe ne peut apparaître comme une machine unique que si chacune de ses ressources apparaît comme une ressource unique et partagée. Pour obtenir un véritable SSI, on pourrait imaginer l'utilisation de plusieurs systèmes existants au sein d'une même grappe afin de globaliser chacune des ressources. Cette solution simpliste n'est cependant pas envisageable. Les systèmes proposés jusqu'à présent sont très complexes et nécessitent la mise en œuvre de mécanismes lourds ou des modifications profondes du système d'exploitation. L'intégration de plusieurs systèmes sur une même grappe conduirait à la présence de nombreux mécanismes logiciels redondants (voir tableau 6.1) et à un système d'une complexité très élevée. Une telle complexité rendrait le système difficile à maintenir et peu fiable. De plus, chaque système a été conçu pour gérer une ressource particulière, sans tenir compte de la présence éventuelle d'autres systèmes. Ainsi, chaque système peut prendre des décisions locales qui peuvent interférer avec les décisions prises par les autres systèmes.

Système considéré	Déplacement de données		Gestion des données dupliquées	
	Migration de pages	Injection de pages	Maintien de la cohérence	Répertoire de localisation
Mosix	Oui	-	-	-
xFS	Oui	-	Oui	Oui
GMS	Oui	Oui	-	Oui
Ivy	Oui	-	Oui	Oui
Koan	Oui	Oui	Oui	Oui

TAB. 6.1 – Comparaison des mécanismes logiciels utilisés dans les systèmes assurant la gestion globale d'une ressource

6.1 Vers un véritable système à image unique

Depuis quelques années, on assiste à l'émergence de systèmes alliant la gestion de plusieurs ressources. Sprite [76] est certainement l'un des premiers systèmes à avoir intégré au sein d'un même système la gestion de plusieurs ressources, à savoir les ressources disque et processeur. Sprite est un système d'exploitation écrit de zéro et intégrant un SGFD assurant une totale transparence de la localisation des fichiers et un système de migration de processus. L'avantage d'allier ces deux mécanismes dans un même système est d'offrir une meilleure transparence de la distribution des ressources et de limiter les contraintes rencontrées à la frontière entre une ressource gérée globalement et une ressource gérée localement. Cependant, Sprite n'offre aucun mécanisme de gestion globale de la mémoire, ne permet pas la migration de *threads* et dispose d'un mécanisme de gestion des caches de fichiers distribués peu performant.

Le système Millipede [41] est un système mis en œuvre au niveau utilisateur sur un système d'exploitation Windows NT, alliant gestion globale de la mémoire et des processeurs. Tout comme dans le système Sprite, cette approche permet une plus grande transparence de la distribution des ressources et limite les contraintes rencontrées à la frontière entre la gestion des processeurs et de la mémoire. Il est en effet possible dans le système Millipede de migrer des *threads*, grâce à la présence d'un mécanisme de MPRL, ce qui permet notamment d'améliorer les performances globales du système. Les *threads* d'une application parallèle peuvent être placés sur des nœuds différents d'une grappe et ainsi augmenter le degré de parallélisme. De manière similaire, lorsqu'une situation de faux partage est détectée entre deux *threads* s'exécutant sur des nœuds distincts, les *threads* sont migrés sur un même nœud. Cette solution diminue certes le parallélisme, mais diminue également le surcoût important introduit par le phénomène de *ping-pong*. Néanmoins, la mise en œuvre de niveau utilisateur de Millipede, introduit de nombreuses contraintes de programmation et limite ses performances. La gestion des données partagées doit par exemple être gérée par le programmeur grâce à un ensemble de fonction spécialisées. D'autre part, l'utilisation d'appels systèmes est fortement limitée. Enfin, le système Millipede ne peut modifier les mécanismes de cache de fichiers, de remplacement ou encore de gestion disque du système

Windows NT sous-jacent.

Le système Genesis [45] est proche du système Millipede, au sens où il vise à offrir un système à image unique fondé sur une gestion globale de la mémoire et des processus. L'originalité de ce système est d'offrir au programmeur le choix d'utiliser un modèle de programmation par mémoire partagée et/ou par échange de messages via une interface PVM. Ce système ayant été mis en œuvre de zéro au niveau noyau, il offre des services plus efficaces et plus transparent à l'utilisation que Millipede.

Nomad [79] est un système ayant pour objectif de réaliser un SSI fondé sur la gestion globale de toutes les ressources d'une grappe : mémoires, disques et processeurs. Ce système inclut des mécanismes de placement et de migration de processus ainsi qu'un SGFD et un SGFP. De plus, Nomad inclut des mécanismes de tolérance aux fautes tels qu'un SGFD fondé sur un stockage redondant des données sur disques et un système de reprise de processus. Cependant, aucun mécanisme de gestion mémoire n'a été proposé et il existe très peu d'informations concernant la conception, la mise en œuvre et les performances de ce système. De plus, le projet Nomad s'est récemment réorienté vers l'étude de mécanismes de gestion de la consommation d'énergie électrique dans une grappe [80].

PM^2[71] est un environnement de programmation distribué à haute performance mis en œuvre au niveau utilisateur. PM^2 repose sur une bibliothèque de *threads* appelée Marcel [31], une couche de communication à haute performance appelée Madeleine [15] et un mécanisme de mémoire partagée appelée DSM-PM^2 [8]. L'objectif de l'environnement PM^2 n'est pas d'offrir un système à image unique mais un environnement de programmation permettant de développer des applications parallèles à haute performance sur une architecture distribuée. Néanmoins, certains mécanismes mis en œuvre au sein de PM^2 tels que la migration de *threads* ou la mémoire virtuelle partagée remplissent les objectifs de transparence que nous avons défini.

Le tableau 6.2 résume les différentes propositions de système à image unique assurant la gestion d'une ou plusieurs ressources, que nous avons évoqué dans cette première partie. Pour chaque type de ressource (mémoire, disque, cache de fichiers et processeur) nous présentons si les propriétés de transparence de localisation ($T.L.$), d'indépendance de localisation ($I.L.$), de partage logique ($P.L.$), de partage physique ($P.P.$) et de transparence à l'utilisation ($T.U.$) ont été réalisées avec succès dans les systèmes considérés.

A la lumière du tableau 6.2, il apparaît qu'il n'existe à ce jour aucun système assurant une gestion globale de toutes les ressources d'une grappe, permettant d'offrir véritablement la vision d'une machine unique au dessus d'une grappe. Nous présentons dans la deuxième partie de ce document la conception d'un mécanisme permettant de gérer globalement toutes les ressources d'une grappe et pouvant s'intégrer très simplement dans un système d'exploitation existant.

	Gestion mémoire					Gestion disque					Cache		Gestion processeur				
	T.U.	T.L.	I.L.	P.P.	P.L.	T.U.	T.L.	I.L.	P.P.	P.L.	P.P.	P.L.	T.U.	T.L.	I.L.	P.P.	P.L.
GMS	Oui	Oui	Oui	Oui	-	-	-	-	-	-	Oui	-	-	-	-	-	-
Ivy	Non[a]	Oui	Oui	-	Oui	-	-	-	-	-	-	-	-	-	-	-	-
Koan	Non[a]	Oui	Oui	Oui[1]	Oui	-	-	-	-	-	-	-	-	-	-	-	-
SciOS	Non[a]	Oui	Oui	-	Oui	-	-	-	-	-	-	-	-	-	-	-	-
NFS	-	-	-	-	-	Oui	Inc	-	Inc[4]	Oui	-	-	-	-	-	-	-
xFS	-	-	-	-	-	Oui	Oui	Oui	Oui	Oui	Oui	Oui	-	-	-	-	-
Mosix	Oui	Inc[2]	-	Inc[2]	-	-	-	-	-	-	-	-[5]	Oui	Oui	Oui	Oui	-
Sprite	-	-	-	-	-	Oui	Oui	Oui	Oui	Oui	-	-	Oui	Oui	Oui	Oui	-
Millipede	Non[a]	Oui	Oui	-	Oui	-	-	-	-	-	-	-	Non[ab]	Oui	Inc[3]	-	Oui
Genesis	Non[a]	Oui	Oui	-	Oui	-	-	-	-	-	-	-	Non[a]	Oui	Oui	Oui	Oui
Nomad	-	-	-	-	-	Oui	Oui	Oui	Oui	Oui	-	-	Oui	Oui	Oui	Oui	Oui
PM2	Non	Oui	Oui	Non	Oui	-	-	-	-	-	-	-	Non	Oui	Inc[6]	-	Oui

Inc : incomplet.
T.U. : Transparence à l'utilisation.
T.L. : Transparence à la localisation.
I.L. : Indépendance à la localisation.
P.P. : Partage physique.
P.L. : Partage logique.
[a] L'application utilisant ce système doit être modifiée et recompilée.
[b] L'utilisation de ce système impose de nombreuses restrictions d'utilisation.
[1] Pas de remplacement sur disque.
[2] Memory ushering.
[3] Pas d'appels systèmes, pas d'utilisation de bibliothèques dynamiques, placement explicite des données partagée en MPRL.
[4] Partage géré explicitement par l'administrateur via les points de montage.
[5] Aucune donnée n'est placée en cache local lors de l'accès à un fichier distant.
[6] Un *thread* ne peut pas être migré s'il effectue des accès aux ressources locales d'une machine, notamment des accès disque.

TAB. 6.2 – Comparaison des propriétés offertes par différents systèmes de gestion globale des ressources mémoire, disque, caches de fichiers distribués et processeur.

Deuxième partie

Conception d'un système à image unique fondé sur une gestion globale de la mémoire physique d'une grappe

7 LES CONTENEURS : POUR UN VÉRITABLE SYSTÈME À IMAGE UNIQUE

7.1 Objectif

L'objectif de cette thèse est de concevoir un système à image unique assurant la gestion de toutes les ressources d'une grappe afin de donner la vision d'une machine unique. Nous pensons que le modèle de programmation par mémoire partagée (comme par exemple OpenMP ou la programmation par *threads*) est plus simple pour la conception d'applications parallèles que les modèles par échange de messages (comme par exemple PVM ou MPI). Son intégration au sein des grappes de calculateurs permettrait de simplifier leur utilisation et ainsi de démocratiser ce type d'architecture. C'est pourquoi nous souhaitons offrir l'image d'une machine multiprocesseur à mémoire partagée de type SMP au dessus d'une grappe. Notre système doit permettre d'exécuter aussi bien des applications séquentielles que des applications parallèles programmées grâce à un modèle de mémoire partagée. Une application multithreadée fondée sur un modèle de mémoire partagée conçue pour un calculateur de type SMP doit pouvoir être exécutée sur notre système sans aucune modification (propriété de transparence à l'utilisation). Enfin, notre système doit permettre de tirer pleinement parti des performances potentielles de l'architecture sous-jacente. Les applications cibles de notre système sont des applications haute performance nécessitant une puissance de calcul élevée et la possibilité de manipuler de gros volumes de données.

Notre système doit offrir une gestion globale de toutes les ressources distribuées de la grappe : processeurs, mémoires et disques et assurer les propriétés présentées au paragraphe 2.3.2 :

- **transparence de localisation** : l'utilisateur ne doit pas avoir à se soucier de la localisation de ses ressources logiques dans la grappe.
- **indépendance à la localisation** : les ressources logiques d'un utilisateur doivent pouvoir être déplacées d'un nœud vers un autre sans que cela ne soit perceptible pour celui-ci.
- **partage logique** : les ressources logiques présentes sur la grappe doivent pouvoir être partagées par les différents processus s'exécutant sur la grappe, quelque soit leur nœud d'exécution.
- **partage physique** : une application gourmande en ressources doit pouvoir tirer bénéfice de toutes les ressources de la grappe.

7.2 Approche proposée

Nous proposons d'offrir un système à image unique fondé sur des mécanismes de niveau système d'exploitation. A ce niveau, il est possible de contrôler toutes les interactions entre l'utilisateur et les ressources matérielles. On peut de cette manière contrôler la répartition des ressources logiques sur l'ensemble des ressources physiques de la grappe et ainsi masquer totalement l'aspect distribué de la grappe à l'utilisateur. De plus, la propriété de transparence à l'utilisation ne peut être réalisée que grâce à une mise en œuvre de niveau noyau.

Cependant, la conception d'un nouveau système d'exploitation dédié aux grappes est une tâche longue et complexe. C'est pourquoi nous proposons la modification d'un système d'exploitation existant, que nous appelons **système hôte**, plutôt que la réalisation complète d'un nouveau système. Cette solution a de nombreux avantages. En premier lieu, la réutilisation d'un système existant accélère et simplifie considérablement la conception et la mise en œuvre. Ensuite, l'évolution du logiciel est assurée naturellement par l'évolution du système hôte. Enfin, la modification d'un système d'exploitation déjà largement utilisé permet une diffusion et une pérennité accrues.

Les précédents travaux visant à offrir un SSI fondé sur la gestion globale d'une seule ressource utilisent des mécanismes spécifiques à chaque ressource, dont la mise en œuvre est très complexe. Multiplier ces mécanismes au sein d'un unique système afin d'assurer la gestion de toutes les ressources risque de conduire à un système complexe, difficile à mettre au point et à maintenir. C'est pourquoi nous proposons de factoriser les composants logiciels utilisés par ces différents mécanismes afin d'obtenir un système plus cohérent, plus simple à réaliser et à maintenir.

Si l'on observe un système d'exploitation, on peut le découper grossièrement en deux parties : (1) les services systèmes de haut niveau et (2) les gestionnaires de périphériques (voir figure 7.1). Les services systèmes offrent une virtualisation des ressources physiques (mémoire, disque, processeur) en fournissant un ensemble de ressources logiques (mémoire virtuelle, fichiers, processus) plus facile à appréhender et à manipuler pour l'utilisateur. Les gestionnaires de périphériques gèrent les ressources à très bas niveau (mot mémoire, secteur de disque dur) en fournissant des abstractions logicielles (page mémoire, bloc disque logique) aux services systèmes.

Dans une grappe, chaque nœud dispose de son propre système d'exploitation composé de ces deux niveaux d'abstraction. Si l'on désire garder la même vision d'une machine unique au dessus d'une grappe, nous ne devons pas modifier les services systèmes de haut niveau. Cependant, ces services doivent pouvoir utiliser et partager l'ensemble des ressources de la grappe. Pour atteindre cet objectif, nous utilisons un service générique intercalé entre les services systèmes et les gestionnaires de périphériques. De la sorte, les services système de haut niveau peuvent tirer profit de l'ensemble des ressources matérielles de la grappe gérées localement par les services de bas niveau.

Notre étude apporte une solution en deux étapes : (1) la définition d'un mécanisme générique de gestion globale des ressources et (2) son intégration au sein d'un système d'exploitation existant.

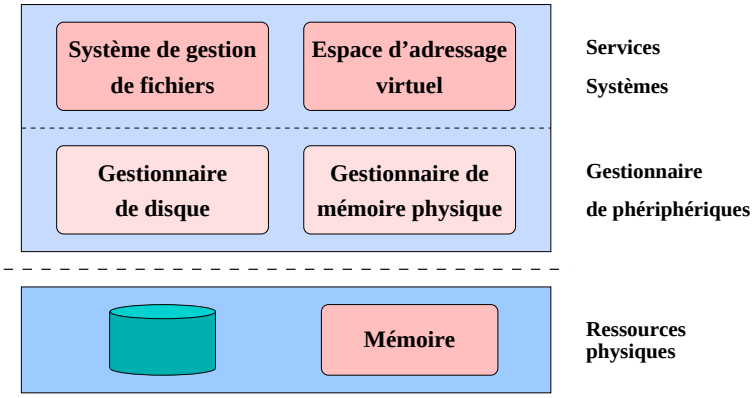

FIG. 7.1 – *Architecture d'un système d'exploitation*

7.3 Mécanisme générique de gestion globale des ressources

Si l'on détaille la conception interne des systèmes d'exploitation, on peut constater une grande homogénéité des mécanismes de gestion des ressources. Les périphériques sont divisés en deux catégories : les périphériques de type caractère (clavier, imprimante, etc) et les périphériques de type bloc (mémoire, disque, etc). Tous les périphériques de type bloc sont gérés suivant une granularité correspondant à une page de mémoire physique : l'unité de gestion de la mémoire virtuelle est la page mémoire, l'unité d'accès aux disques et de gestion du cache de fichiers est la page mémoire.

Au sein d'un système d'exploitation, la page mémoire est utilisée comme unité de gestion et de partage des données entre les différents niveaux du système (du matériel aux processus). Il semble donc judicieux d'utiliser la page mémoire comme unité de gestion et de partage de données entre les différents systèmes d'exploitation s'exécutant sur les nœuds de la grappe. Nous appelons **partage vertical** le partage de pages entre les différents niveaux du système d'exploitation et **partage horizontal** le partage entre les différents nœuds de la grappe (voir figure 7.2).

Dans un système centralisé, les pages physiques utilisées par un processus sont structurées en unités cohérentes au sein de son espace d'adressage grâce à des segments de mémoire virtuelle. Nous proposons d'étendre ce concept à l'échelle d'une grappe au sein d'un système d'exploitation distribué, grâce au concept de **conteneur**. Un conteneur est un objet permettant de stocker et d'échanger des pages entre les nœuds d'une grappe. A chaque conteneur est associée une sémantique permettant d'identifier les données qu'il contient et les méthodes d'accès à celui-ci. Enfin, un conteneur est global à la grappe. Grâce aux conteneurs, nous assurons le partage horizontal. Les pages contenues dans les conteneurs peuvent être partagées et accédées par tous les nœuds d'une grappe. En intégrant ce mécanisme générique de partage au sein du système hôte, il est possible de lui donner l'illusion qu'il dispose d'une mémoire physiquement partagée au dessus de laquelle il est possible de mettre en œuvre facilement des services systèmes distribués (voir figure 7.2).

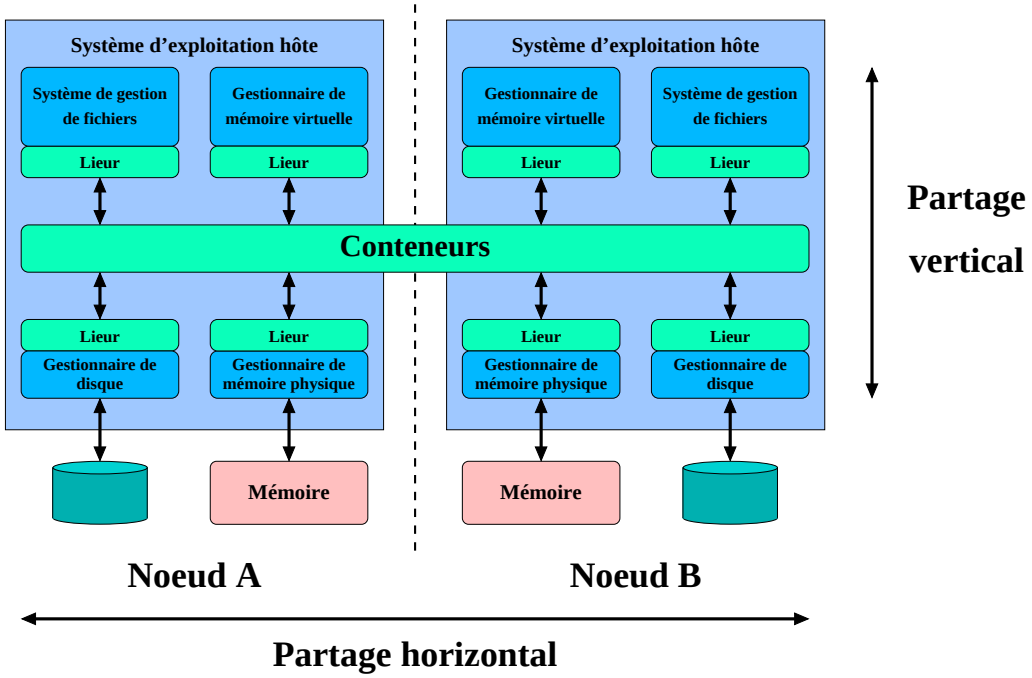

FIG. 7.2 – *Gestion globale des ressources d'une grappe grâce au concept de conteneur*

L'utilisation d'un mécanisme unique pour la gestion globale de l'ensemble des ressources du système possède de nombreux avantages. En premier lieu, le code nécessaire à sa mise en œuvre est de taille réduite, ce qui le rend sa maintenance aisée et permet de le fiabiliser. Ensuite, les décisions concernant la gestion des pages est réalisée par une seule entité logicielle disposant d'informations globales sur l'état de la grappe. On évite ainsi les décisions conflictuelles qui peuvent apparaître lors de l'utilisation de différents mécanismes. De plus, l'amélioration de ce mécanisme, en terme de performance, de robustesse ou de tolérance au fautes, améliore l'ensemble des services systèmes distribués.

7.4 Conception de services de haut niveau sur conteneurs

Un conteneur est un objet global permettant la gestion et le partage de pages entre les nœuds d'une grappe. Pour permettre l'utilisation des conteneurs par les services de haut niveau, il est nécessaire de concevoir un mécanisme permettant de réaliser la liaison entre les conteneurs et le système hôte. Dans ce but, nous proposons la notion de **lieurs**. Un lieur est un élément logiciel très court permettant de réaliser la connexion entre les conteneurs et le système d'exploitation hôte. A chaque conteneur est associé une paire de lieurs : un lieur de haut niveau appelé **lieur d'interface** permettant de détourner les fonctions d'accès aux périphériques vers le conteneur et un lieur de bas niveau appelé **lieur d'entrée/sortie** permettant au conteneur d'accéder à un gestionnaire de périphérique. Un conteneur donné ne peut être lié qu'à un seul gestionnaire de périphérique. En revanche, il peut être lié à

plusieurs services système de haut niveau. Les lieurs permettent ainsi de réaliser le partage vertical des données à travers les conteneurs (voir figure 7.2).

Les lieurs sont utilisés afin de réaliser très simplement un grand nombre de services systèmes distribués. Pour chaque service système distribué de haut niveau, une paire de lieurs différente est utilisée. Par exemple, une paire de lieurs permettant de connecter un conteneur entre le gestionnaire de mémoire virtuelle et le gestionnaire de mémoire physique permet de réaliser une mémoire virtuelle partagée. Une paire de lieurs permettant de connecter un conteneur entre le système de gestion de fichiers virtuel et le gestionnaire de disque permet de réaliser un système de gestion de fichiers distribué et un cache coopératif

Un processus est trop intimement lié au système d'exploitation pour pouvoir bénéficier directement du mécanisme de conteneur. Cependant, un processus utilise de nombreux services système (mémoire virtuelle, gestion de fichier, etc) qui peuvent être virtualisés à l'échelle de la grappe grâce aux conteneurs. Dans ce contexte, la migration de processus est amplement simplifiée puisque la gestion de l'accès aux ressources distantes est assurée naturellement par les conteneurs.

8 CONCEPTION D'UN MÉCANISME GÉNÉRIQUE DE GESTION DES RESSOURCES : LES CONTENEURS

8.1 Cadre de l'étude

Nous nous intéressons dans cette étude à la conception d'un mécanisme logiciel permettant d'offrir l'illusion d'une machine unique au dessus d'une grappe de calculateurs. Nous considérons une grappe de machines mono-processeurs interconnectées par un réseau à haut débit et disposant d'un nombre limité de nœuds (moins de 128). Pour des raisons de distribution des services, de bande passante et de latence réseau, le passage à l'échelle du système que nous souhaitons concevoir nécessite la conception de mécanismes spécialisés qui ne sont pas étudiés dans ce document.

Nous supposons que les nœuds de la grappe exécutent un système d'exploitation hôte de type Unix™. Au sein de ce système, l'unité d'exécution est le processus tel qu'il a été définit dans la partie A.3.1. Au sein d'un processus peuvent s'exécuter plusieurs *threads* partageant les données globales et les fichiers ouverts manipulés par ce processus. La mise en œuvre des *threads* étant très dépendante du système d'exploitation hôte, nous faisons l'hypothèse simplificatrice qu'il est possible de migrer un *thread* de la même manière qu'un processus. Les détails de mise en œuvre seront abordés dans le chapitre 10. La gestion des ressources de type bloc (mémoire, disques) est réalisée suivant une granularité de l'ordre de la page mémoire : allocation de mémoire physique, de mémoire virtuelle, taille des blocs du cache de fichier, taille des accès disques élémentaires, etc.

Le modèle de programmation considéré pour la conception d'applications parallèles s'exécutant sur notre système est la programmation par variables partagées en mémoire grâce à l'utilisation d'un langage spécialisé de type OpenMP ou simplement grâce à des *threads*. Ce modèle est le modèle utilisé sur les machines à mémoire partagée de type SMP. Notre objectif est de pouvoir exécuter ce type d'applications sur une grappe de calculateurs sans aucune modification.

Sur une machine à mémoire partagée, la communication entre les « *threads* » d'une application parallèle est réalisée grâce au partage de variables en mémoire. La mémoire étant physiquement partagée, cette communication ne pose aucun problème. Dans le cas d'une grappe, ce partage est impossible puisque la mémoire est distribuée sur les différents nœuds. Deux « *threads* » s'exécutant sur deux nœuds différents ne peuvent pas communiquer par partage de variables en mémoire.

Nous présentons dans la suite de ce chapitre un mécanisme logiciel appelé **conte-**

neur [83] qui permet de partager la mémoire physique des nœuds d'une grappe au plus bas niveau d'un système d'exploitation. Ce mécanisme est intégré au sein d'un système d'exploitation hôte grâce à des **lieurs** qui permettent de réaliser très simplement des services systèmes distribués de haut niveau tel qu'une mémoire virtuelle partagée, un cache coopératif ou encore un système de gestion de fichiers distribué. L'objectif de ces mécanismes est d'offrir la vision d'une machine SMP aux applications s'exécutant sur une grappe.

8.2 Conteneur : définition

Définition 23 (Conteneur) *Un **conteneur** est une entité virtuelle permettant de stocker et de partager des données au sein d'une architecture distribuée pour le compte d'un ensemble de noyaux de système d'exploitation.*

Un conteneur est constitué d'un ensemble de pages mémoires qui peuvent être partagées entre les différents nœuds d'une grappe par les noyaux des systèmes d'exploitation hôtes s'y exécutant. Les données stockées dans un conteneur sont placées dans des cadres de page de la mémoire physique des nœuds, pouvant être manipulés par le système hôte comme des cadres de page ordinaires. Ceux-ci peuvent être projetés dans l'espace d'adressage d'un processus, constituer un bloc dans le cache de fichier, etc. Lorsqu'une page d'un conteneur est présente en mémoire locale, les données qu'elle contient sont accessibles par les opérations de lecture et écriture du processeur.

Le système hôte n'a pas connaissance de l'existence des conteneurs. Ceux-ci sont intégrés dans le noyau grâce à des lieurs, qui sont des éléments logiciels intercalés dans le noyau entre les gestionnaires de périphériques et les services systèmes (voir figure 8.1). De nombreux mécanismes du noyau reposent sur la manipulation de pages physiques. Les lieurs détournent ces mécanismes pour assurer le partage des données à travers les conteneurs.

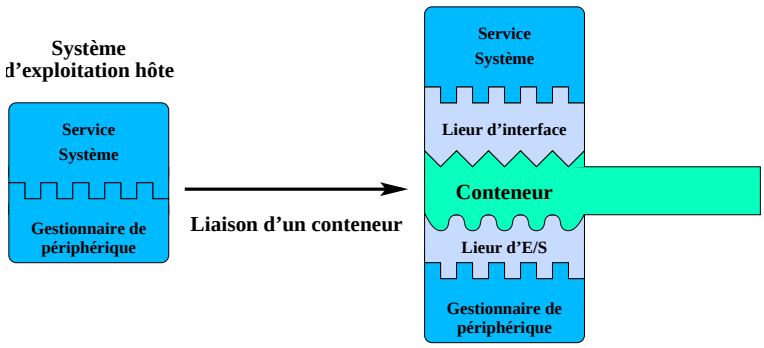

FIG. 8.1 – *Intégration d'un conteneur dans le système hôte grâce aux lieurs*

Lors de la création d'un nouveau conteneur, un lieur d'entrée/sortie lui est associé. Le conteneur cesse alors d'être un objet générique et permet de partager des données provenant du périphérique auquel il est lié. On dit que l'on a instancié le conteneur. L'instanciation

d'un conteneur permet de lui donner une sémantique identifiant la nature des données qu'il contient. On utilise donc autant de conteneurs que de données sémantiquement différentes que l'on désire partager. Par exemple, un conteneur différent est utilisé pour chaque fichier et un conteneur différent pour chaque segment de mémoire partagée.

Les services systèmes sont connectés aux conteneurs grâce à des lieurs d'interface. Il est possible de connecter plusieurs services systèmes sur le même conteneur. Ainsi, on peut par exemple projeter un conteneur dans l'espace d'adressage d'un processus A et y accéder grâce à une interface de fichier au sein d'un processus B.

8.2.1 Gestion du partage

Au sein d'un conteneur, le partage des données entre les différents nœuds de l'architecture est assuré grâce à un mécanisme de partage de mémoire comparable à une MPRL[1]. Le mécanisme de conteneur utilise la mémoire physique des nœuds d'une grappe comme un cache pour les pages des conteneurs, ceci à la manière des architectures COMA [47]. Lorsqu'une page est référencée sur un nœud, si celle-ci n'est pas présente localement un exemplaire provenant d'un nœud distant disposant d'une copie est placé en mémoire locale. Il peut donc exister plusieurs copies d'une même page dans la grappe. La cohérence de ces copies est assurée grâce à un protocole de cohérence par invalidation sur écriture (voir paragraphe 3.2.3).

A ce titre, un conteneur est très proche du concept de MPRL, puisque ces deux mécanismes disposent de deux importantes caractéristiques communes : le partage des données est assuré grâce à un mécanisme logiciel de partage de mémoire et la cohérence des données dupliquées est assurée grâce à un protocole de cohérence. Cependant, à la différence des MPRL, un conteneur réalise une gestion de la mémoire physique au plus bas niveau d'un système d'exploitation. Les données partagées par un conteneur le sont pour le compte des noyaux de système d'exploitation hôte et non pas pour le compte des utilisateurs. Bien entendu, la plupart des données manipulées par le noyau sont généralement des données appartenant à des processus utilisateurs : segment de données mémoire, fichier, etc. Néanmoins, un conteneur peut être totalement dissocié de tout contexte de processus. Par exemple lorsqu'un conteneur contient un fichier placé en cache, il peut exister et fonctionner sans être rattaché à un quelconque processus et donc à un utilisateur.

8.2.2 Interface d'accès aux conteneurs

Si l'on peut comparer un conteneur à une MPRL pour la gestion du partage et de la cohérence des données, le mécanisme d'accès et de gestion des pages d'un conteneur est totalement nouveau. Les méthodes classiques d'accès aux données partagées d'une MPRL ne sont pas applicables ici. Le détournement de la mémoire virtuelle n'a aucun sens au sein d'un système d'exploitation puisque la plupart d'entre eux fonctionnent en manipulant directement des adresses physiques. L'instrumentation est difficilement envisageable puisque instrumenter tout le noyau du système hôte représente une tâche considérable. De plus, la

[1] Mémoire Partagée Répartie Logicielle (voir paragraphe 3.2.1)

plupart des données placées en conteneurs sont utilisées par les processus utilisateurs, qu'il faudrait donc également instrumenter.

L'accès et la manipulation des conteneurs sont donc réalisés grâce à un ensemble de fonctions spécialisées représentant l'interface d'accès aux conteneurs. Cette interface est utilisée par les lieurs afin d'assurer aux services de haut niveau l'accès aux données des conteneurs tout en assurant la cohérence de celles-ci. L'interface est composée des quatre fonctions détaillées ci-après.

8.2.2.1 find_page

La fonction *find_page* permet de vérifier la présence d'une page particulière d'un conteneur en mémoire locale. Si la page demandée est présente en mémoire locale, son adresse physique est retournée et peut être utilisée par le lieur et le système hôte. Si la page n'est pas présente localement, un code d'erreur est retourné.

8.2.2.2 get_page

La fonction *get_page* permet d'obtenir une copie d'une page particulière d'un conteneur dans la mémoire locale. Si la page demandée n'est pas présente dans la mémoire locale, une copie provenant d'un nœud disposant d'une copie est placée en mémoire centrale. Si aucun nœud ne dispose d'une copie, une nouvelle page est créée. Dans tous les cas, l'adresse physique de la page en mémoire locale est retournée. Cette fonction peut être rapprochée du mécanisme de défaut de page en lecture d'une MPRL.

8.2.2.3 grab_page

A la manière de la fonction *get_page*, la fonction *grab_page* permet d'obtenir une copie d'une page particulière d'un conteneur dans la mémoire locale. Cependant, à la différence de la fonction *get_page*, la fonction *grab_page* assure que la page placée dans la mémoire locale est la seule copie existant dans la grappe. Cette fonction doit être utilisée avant de réaliser un accès en écriture sur une page d'un conteneur. On peut rapprocher cette fonction du mécanisme de défaut en écriture d'une MPRL.

8.2.2.4 flush_page

La fonction *flush_page* permet de supprimer une page d'un conteneur de la mémoire locale. Cette fonction est utilisée afin de libérer des cadres de pages lorsque la mémoire locale est saturée. S'il existe d'autres exemplaires de la page sur des nœuds distants, la copie locale est simplement détruite. Si la copie locale est la seule copie existante, elle est injectée dans la mémoire d'un nœud distant. Si aucun nœud ne dispose de place mémoire pour réaliser cette injection, la page est remplacée sur disque.

8.2.3 Intégration des conteneurs dans le système hôte : les lieurs

Définition 24 (Lieur) *Un **lieur** est un élément logiciel inséré dans le noyau du système hôte afin d'assurer la connexion entre les noyaux et les conteneurs.*

Les conteneurs sont introduits dans un système hôte par l'intermédiaire des **lieurs**. Les lieurs sont insérés dans le noyau du système hôte entre les mécanismes assurant les services systèmes de haut niveau (gestion de la mémoire virtuelle, système de fichiers, cache de fichiers, etc) et les gestionnaires de périphériques (gestionnaire de mémoire physique, système d'entrée/sortie de bas niveau, etc).

Nous distinguons deux types de lieurs : (1) les **lieurs d'interface**, permettant de détourner les accès aux mécanismes de gestion des périphériques et de les rediriger vers les conteneurs et (2) les **lieurs d'entrée/sortie**, assurant l'échange des données entre les conteneurs et les périphériques.

8.2.3.1 Connexion des conteneurs aux services systèmes

Définition 25 (Lieur d'interface) *Un **lieur d'interface** est un lieur permettant de faire coïncider l'interface des conteneurs à l'interface des gestionnaires de périphériques utilisée par les services systèmes.*

L'interface de base des conteneurs n'a pas été conçue pour être utilisée directement par le noyau, mais par les lieurs d'interface. Un lieur d'interface permet de changer l'interface d'un conteneur et de l'adapter aux exigences des services systèmes de haut niveau. Cette interface doit donner l'illusion au noyau qu'il communique avec les gestionnaires de périphériques avec lesquels il interagit traditionnellement (voir figure 8.1). Il est ainsi possible de "tromper" le noyau et de détourner les accès aux périphériques vers les conteneurs. Pour cela, le lieur d'interface doit s'assurer qu'il fait coïncider les fonctions de gestion des conteneurs aux fonctions de l'interface qu'il propose aux services de haut niveau.

La modification des droits d'accès aux pages d'un conteneur peut avoir des répercutions sur les services systèmes de plus haut niveau. Par exemple, l'invalidation d'une page d'un conteneur projeté en mémoire virtuelle entraîne son invalidation dans l'espace d'adressage correspondant. Pour cela, chaque lieur d'interface dispose d'une fonction appelée par les conteneurs lors de la modification de l'état d'une page :

change_access : cette fonction permet de refléter les modifications de l'état d'une page d'un conteneur sur l'interface d'accès à cette page fournie par le lieur.

8.2.3.2 Entrée/Sortie de données en conteneur

Définition 26 (Lieur d'entrée/sortie) *Un **lieur d'entrée/sortie** est un lieur permettant d'assurer l'échange de données entre les conteneurs et le gestionnaire d'un périphérique donné.*

Juste après la création d'un conteneur, celui-ci est totalement vide, c'est-à-dire qu'il ne contient aucune page et qu'aucun cadre de page ne contient de données du conteneur. Les pages sont créées et les cadres de pages alloués à la demande lors du premier accès à une page. De la même manière, des données peuvent être supprimées d'un conteneur lors de la destruction de celui-ci ou pour libérer temporairement des cadres de page lorsque la mémoire de la grappe est saturée.

Les mécanismes mis en jeu lors de l'entrée et de la sortie de données d'un conteneur sont fortement dépendants du type de données qu'il contient. Prenons l'exemple d'un conteneur stockant les données d'un segment de l'image mémoire d'un processus. Lors d'un premier accès, il suffit d'allouer une nouvelle page physique et de l'insérer dans le conteneur. Lors de l'éviction d'une page, celle-ci doit être placée dans la zone d'échange du disque afin de pouvoir qu'on puisse y accéder à nouveau dans le futur. Considérons maintenant l'exemple d'un conteneur stockant les données d'un fichier. Lors d'un premier accès à une page, les données du fichier correspondant à cette page doivent être chargées depuis l'unité de stockage et placées dans un nouveau cadre de page. Cette page pourra alors être placée dans le conteneur. Lors de l'éviction d'une page, celle-ci doit être recopiée dans le fichier associé au conteneur si elle a été modifiée depuis son chargement.

Ces opérations dépendantes du type de données manipulées par un conteneur sont réalisées par les **lieurs d'entrée/sortie**. Pour chaque type de données, un lieur différent est réalisé afin d'assurer les fonctions de premier accès et d'éviction. Lorsqu'un premier accès ou qu'une éviction a lieu sur une page d'un conteneur, la fonction correspondante du lieur associé au conteneur est appelée.

Un conteneur est par définition un objet générique, pouvant contenir tout type de données. Lorsque l'on associe un lieur d'entrée/sortie à un conteneur, on lui donne une sémantique. Le conteneur héberge alors des données provenant d'un type particulier de périphérique. On dit alors que l'on a **instancié** le conteneur.

Un lieur d'entrée/sortie doit fournir les fonctions d'interface suivantes :

first_touch : cette fonction permet d'allouer un cadre de page lors du premier accès à une page d'un conteneur. Si le conteneur hébergeant la page est associé à un périphérique de stockage, la donnée doit être chargée depuis ce périphérique dans le cadre de page alloué.

flush_page : cette fonction permet d'évincer un cadre de page de la mémoire physique de la grappe. Cette fonction est appelée par un conteneur lorsqu'une page a été choisie pour être remplacée. Cette page pourra éventuellement être rechargée plus tard dans le conteneur. Si le conteneur hébergeant la page est associé à un périphérique de stockage, celui-ci doit être mis à jour, sinon la page doit être placée dans un espace de stockage temporaire.

free_page : cette fonction permet de libérer un cadre de page de la mémoire physique de la grappe. Cette fonction est appelée par un conteneur lorsqu'une page est définitivement supprimée d'un conteneur. Cette situation survient notamment lors de la destruction d'un conteneur. Si le conteneur hébergeant la page est associé à un périphérique de stockage, celui-ci doit être mis à jour.

invalidate_page : cette fonction permet de libérer les ressources liées à une page lorsque celle-ci doit être invalidée pour le respect du protocole de cohérence. Lors d'un appel à cette fonction, il est inutile de mettre à jour un éventuel périphérique de stockage secondaire.

8.2.4 Notion d'attachement

Définition 27 (Attachement) *Un conteneur est dit* ***attaché*** *lorsque les opérations d'entrée/sortie qui lui sont associées doivent être réalisées sur un nœud particulier de la grappe.*

Lors du premier accès ou de l'éviction d'une page, le nœud sur lequel s'exécute la fonction du lieur d'entrée/sortie associé peut avoir une importance. Le chargement d'une page depuis un fichier doit être réalisé sur le nœud sur lequel le fichier est stocké, alors que la création d'une nouvelle page pour un segment de mémoire peut être réalisée sur n'importe quel nœud. De même, l'éviction d'une page contenant les données d'un fichier doit être réalisée sur le nœud de stockage du fichier, alors que l'éviction d'une page associée à un segment de mémoire peut être réalisée sur n'importe quel nœud.

Certains conteneurs sont donc dépendants d'un nœud particulier pour l'exécution des fonctions de premier accès et d'éviction, alors que d'autres en sont totalement indépendant. Les conteneurs dépendant d'un nœud particulier sont les conteneurs stockant des données provenant d'un périphérique situé sur un nœud de la grappe. L'accès à ce périphérique doit être effectué sur le nœud l'hébergeant.

Pour différencier les conteneurs dépendant d'un nœud des conteneurs indépendant d'un nœud, nous utilisons la notion d'*attachement*. Un conteneur est dit *attaché* lorsqu'il contient des données associées à un périphérique localisé sur un nœud particulier de la grappe. Un conteneur est dit *non attaché* lorsqu'il contient des données ne provenant pas d'un périphérique particulier de la grappe.

8.2.5 Gestion du partage de conteneurs

Par définition, un conteneur est partagé par tous les « *threads* » d'une application. En revanche, le partage de conteneurs entre processus n'est réalisé que suite à une demande explicite des lieurs. Un conteneur est une unité indissociable de partage de données, c'est-à-dire que toutes les modifications effectuées sur les données d'un conteneur sont immédiatement visibles par toutes les activités accédant à ce conteneur : « *threads* », processus ou noyau. Un conteneur ne peut donc être partagé entre plusieurs processus que si le conteneur est partagé en lecture seule (par exemple un conteneur hébergeant le texte d'un programme) ou si ces processus ont demandé explicitement à partager un conteneur en lecture et en écriture (par exemple un conteneur utilisé comme segment de mémoire partagée).

Au sein d'un système d'exploitation, un objet (fichier, segment de mémoire, etc) peut être partagée de manière hybride, c'est-à-dire accessible en lecture et en écriture par plusieurs processus, mais les écritures effectuées par un processus ne sont pas visibles par les

autres processus (ces écritures sont cependant visibles par tous les « *threads* » s'exécutant au sein du processus écrivain). Ce cas de figure correspond à la copie paresseuse d'un objet réalisée grâce à un mécanisme de copie sur écriture. Cette solution est largement utilisée afin d'éviter le gaspillage de mémoire physique et les recopies inutiles, notamment pour la copie de l'image d'un processus lors de la création d'un processus fils ou pour la projection privée d'un fichier en mémoire virtuelle.

Ce partage hybride, est réalisé au sein des conteneurs grâce à un mécanisme de copie sur écriture similaire au mécanisme de copie sur écriture utilisé par un système d'exploitation. Il est ainsi possible de partager, avec copie sur écriture, un segment de mémoire entre deux conteneurs suite à la création d'un fils de processus, ou encore de réaliser une projection privée de conteneur hébergeant un fichier en mémoire virtuelle.

8.3 Protocole de gestion de la cohérence séquentielle

Ce paragraphe présente le protocole de gestion de la cohérence utilisé au sein des conteneurs. Nous utilisons un protocole à cohérence séquentielle[63] fondé sur un mécanisme d'invalidation sur écriture[54]. La gestion de la cohérence est assurée grâce aux fonctions *get_page* et *grab_page*.

8.3.1 Notion de propriétaire et de gestionnaire

Nous avons choisi d'assurer la cohérence des données en conteneur grâce à un mécanisme d'invalidation sur écriture : lorsqu'un nœud veut écrire sur une page, toutes les copies de celle-ci sont invalidées au préalable. Avec cette stratégie, un seul nœud est le **propriétaire** d'une page d'un conteneur à un instant donné. Lorsqu'une page n'a jamais été accédée, il n'existe aucune copie dans la grappe. Nous considérons alors qu'elle n'a pas de propriétaire.

Définition 28 (Propriétaire) *Le nœud **propriétaire** d'une page est nœud ayant réalisé la dernière écriture sur la page ou l'accès en lecture ayant provoqué le chargement de la page en conteneur.*

Définition 29 (Copie maître) *La **copie maître** d'une page est la copie située sur le nœud propriétaire.*

Définition 30 (Duplicata) *Toutes les copies d'une page à l'exception de la copie maître sont appelées **duplicata**.*

Lorsqu'un nœud désire obtenir une page d'un conteneur, il doit dialoguer avec le propriétaire de celle-ci. Étant donné que le propriétaire d'une page change au cours du temps, il est nécessaire de mettre en œuvre un protocole permettant de retrouver l'identité du nœud propriétaire d'une page à tout instant.

Pour localiser le propriétaire d'une page, nous utilisons une version modifiée de la gestion distribuée statique du propriétaire proposée par Li [64]. Cette solution offre un bon compromis entre l'efficacité et la facilité de mise en œuvre de la localisation du propriétaire.

Avec cette approche, chaque nœud est le **gestionnaire** d'un sous-ensemble de pages qui lui sont attribuées statiquement grâce à une fonction de distribution. Nous utilisons une fonction *modulo* qui distribue cycliquement la gestion des pages sur les nœuds. Chaque gestionnaire dispose d'une table permettant d'identifier le propriétaire de chaque page dont il a la charge.

De plus, chaque nœud gère une table définissant pour chaque page d'un conteneur l'état des copies locales. Trois états différents sont utilisés : **invalide**, **lecture** et **lecture-écriture**. L'état **invalide** indique que la page n'est pas présente sur le nœud. L'état **lecture** indique que la page est présente sur le nœud mais accessible uniquement en lecture. Enfin, l'état **lecture-écriture** indique que la page est présente sur le nœud et accessible en lecture et en écriture.

Enfin, chaque propriétaire dispose d'une table permettant de connaître l'ensemble des duplicatas existants pour chaque page dont il est propriétaire.

8.3.2 Obtention d'une copie : la fonction get_page

La fonction *get_page* permet d'obtenir une copie en mémoire locale d'une page d'un conteneur. Elle est appelée lors d'un accès à une page qui se trouve dans l'état **invalide**, c'est-à-dire qui n'est pas présente en mémoire locale. Lorsqu'un nœud N fait appel à cette fonction pour obtenir une copie de la page p du conteneur c, une requête de lecture est envoyée au gestionnaire de la page *(c,p)* **[1]** (voir figure 8.2). Le gestionnaire fait suivre la requête au propriétaire de la page **[2]**, qui ajoute l'identifiant de ce nœud dans la liste des duplicatas (*copyset*) **[3]**, place son droit d'accès sur la page *(c,p)* à *lecture* et envoie une copie de cette page au nœud N **[4]**. Le nœud N reçoit la page et place son droit d'accès sur la page *(c,p)* à *lecture* **[5]**. L'algorithme de traitement de la fonction *get_page* ainsi que les actions engendrées sont présentés en annexe B.2.

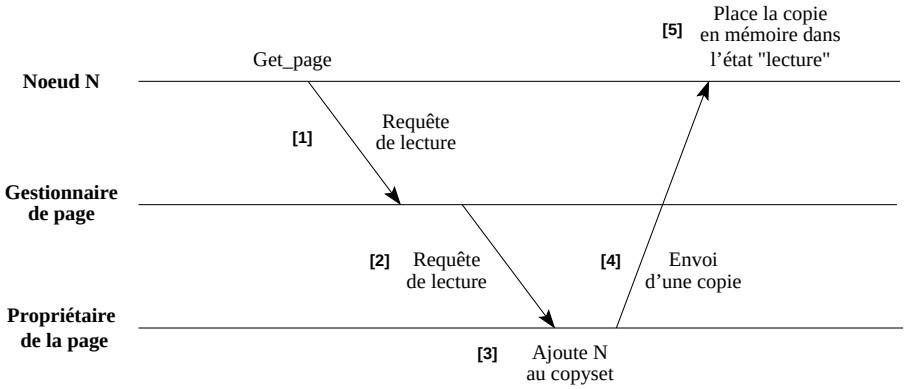

FIG. 8.2 – *Algorithme de traitement de la fonction Get_Page*

8.3.3 Obtention d'une copie unique : la fonction grab_page

La fonction *grab_page* permet d'obtenir une copie en mémoire locale d'une page d'un conteneur et de s'assurer que cette copie est la seule existante dans la grappe. Lorsqu'un nœud N fait appel à cette fonction pour obtenir une copie de la page p du conteneur c, une requête d'écriture est envoyée au gestionnaire de la page *(c,p)* **[1]** (voir figure 8.3). Le gestionnaire fait suivre la requête au propriétaire de la page **[2]** et note que le nouveau propriétaire est désormais le nœud N. Le propriétaire de la page invalide tous les duplicatas existants **[3]**, attend les acquittements **[4]** puis envoie une copie de la page au nœud N **[5]**. Enfin, il invalide sa copie locale. Le nœud N reçoit la page, place son droit d'accès sur la page *(c,p)* à *lecture-écriture* **[6]** et envoie un acquittement au gestionnaire **[7]**. L'algorithme de traitement de la fonction *grab_page* ainsi que les actions engendrées sont présentés en annexe B.3.

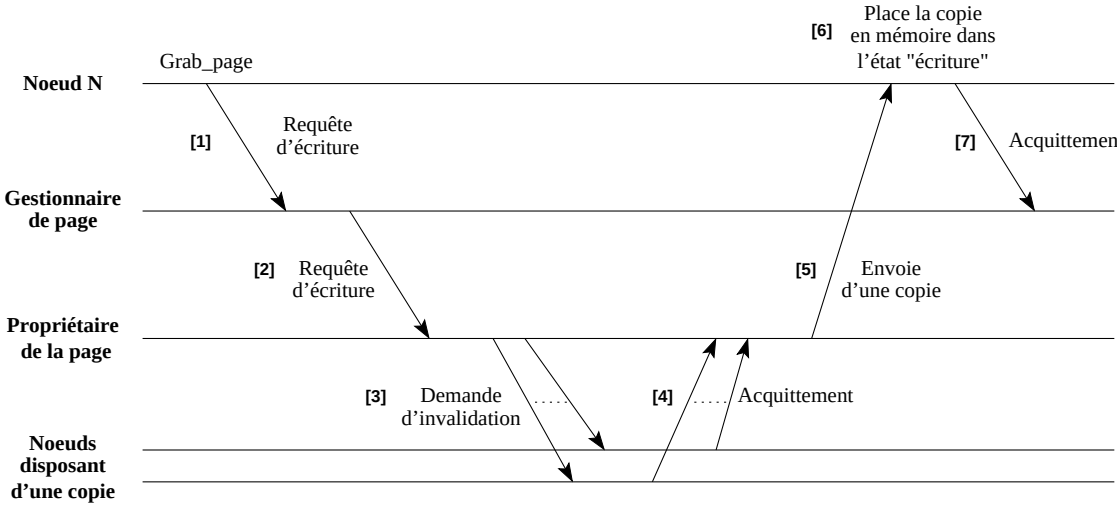

FIG. 8.3 – *Algorithme de traitement de la fonction Grab_Page*

8.4 Entrée des données en conteneur

Lorsqu'un conteneur est créé, il n'existe aucune copie de ses pages dans la mémoire de la grappe. Les pages sont créées à la demande par le conteneur via le lieur d'entrée/sortie associé, lors du premier accès à celles-ci. Deux tables dupliquées sur tous les nœuds permettent de déterminer quel lieur d'entrée/sortie et quels lieurs d'interface sont associés à chaque conteneur. L'appel des fonctions des lieurs d'entrée/sortie pouvant être attaché à un nœud particulier, nous utilisons une table supplémentaire indiquant à quel nœud chaque conteneur est attaché.

Nous présentons dans la suite de cette partie les modifications apportées aux fonctions *get_page* et *grab_page* afin d'intégrer les mécanismes de premier accès et de chargement de données en conteneur.

8.4.1 Modification de la fonction get_page

Lorsque le gestionnaire de la page *(c,p)* reçoit une requête de lecture provenant du nœud N, il vérifie s'il existe une copie de cette page dans la grappe. Si c'est le cas, le protocole décrit précédemment est utilisé. Dans le cas contraire, si le conteneur n'est pas attaché ou si le nœud attaché au conteneur est le nœud N, le gestionnaire note que le propriétaire de la page est le nœud N et retourne un message de création de page locale au nœud N. Celui-ci fait alors appel à la fonction *first_touch* du lieur d'entrée/sortie associé au conteneur afin de charger dans le conteneur un exemplaire de la page. Il place enfin son identifiant dans le *copyset* et indique que la page est accessible en lecture. Enfin, il fait appel à la fonction *change_access* du lieur d'interface afin de répercuter les changement de droit d'accès à la page sur les services systèmes de haut niveau.

Si le conteneur est attaché à un nœud qui n'est pas le nœud N, le gestionnaire note que le propriétaire de la page *(c,p)* est le nœud attaché et envoie une requête de création de page en lecture au nœud attaché. Celui-ci fait alors appel à la fonction *first_touch* du lieur d'entrée/sortie associé au conteneur afin de charger en conteneur un exemplaire de la page. Il place son identifiant, ainsi que celui du nœud N dans le *copyset* et envoie une copie de la page au nœud N. Celui-ci place alors la page en mémoire locale dans l'état lecture et fait appel à la fonction *change_access* du lieur d'interface afin de répercuter les changement de droit d'accès à la page sur les services systèmes de haut niveau. Deux nœuds disposent alors d'une copie de la page en lecture.

La modification de l'algorithme de traitement de la fonction *get_page* ainsi que les actions engendrées par celle-ci sont présentés en annexe B.4.

8.4.2 Modification de la fonction grab_page

Lorsque le gestionnaire de la page *(c,p)* reçoit une requête d'écriture provenant du nœud N, il vérifie s'il existe une copie de cette page dans la grappe. Si c'est le cas, le protocole décrit précédemment est utilisé. Dans le cas contraire, si le conteneur n'est pas attaché ou si le nœud attaché au conteneur est le nœud N, le gestionnaire note que le propriétaire de la page est le nœud N et lui retourne un message de création de page locale. Celui-ci fait alors appel à la fonction *first_touch* du lieur d'entrée/sortie associé au conteneur afin de charger en conteneur un exemplaire de la page. Il place enfin son identifiant dans le *copyset*, indique que la page est accessible en lecture/écriture, fait appel à la fonction *change_access* du lieur d'interface et envoie un acquittement au gestionnaire.

Si le conteneur est attaché à un nœud qui n'est pas le nœud N, il note que le propriétaire de la page *(c,p)* est le nœud N et envoie une requête de création de page en écriture au nœud attaché. Celui-ci fait alors appel à la fonction *first_touch* du lieur d'entrée/sortie associé au conteneur afin de charger en conteneur un exemplaire de la page. Il envoie une copie de la page au nœud N et supprime la copie locale en faisant appel à la fonction *invalide_page* du lieur d'entrée/sortie. Le nœud N place alors son identifiant dans le *copyset*, indique que la page est accessible en lecture/écriture, fait appel à la fonction *change_access* du lieur d'interface et envoie un acquittement au gestionnaire.

La modification de l'algorithme de traitement de la fonction *grab_page* ainsi que les actions engendrées par celle-ci sont présentés en annexe B.5.

8.5 Remplacement de pages : la fonction flush_page

Lorsque la mémoire d'un nœud est saturée, le mécanisme de pagination est activé afin de libérer des cadres de page. Se pose alors le problème de la sélection des pages à évincer. Dans un système d'exploitation centralisé, ce choix est généralement fondé sur une approximation de l'algorithme LRU : à chaque page est associée la date du dernier accès. Lorsqu'une page doit être évincée, la page ayant la date de dernier accès la plus ancienne est choisie. Cette page est alors placée dans la zone d'échange du disque local et supprimée de la mémoire physique.

L'introduction des conteneurs dans le système d'exploitation modifie singulièrement la politique de remplacement. La distribution des pages à travers les mémoires locales et la possibilité d'avoir plusieurs copies d'une même page dans la grappe rend obsolète l'approche centralisée du remplacement. La politique de sélection des pages à remplacer et les mécanismes de remplacement doivent être adaptées à cette nouvelle configuration.

Dans la première partie de ce paragraphe, nous présentons les politiques utilisées dans le cadre des conteneurs pour choisir les pages à remplacer et les nœuds d'injection. Nous présentons ensuite les mécanismes utilisés pour effectuer le remplacement.

8.5.1 Choix de la page à remplacer

La première chose à effectuer lors d'un remplacement de page est de choisir la page à évincer. La politique LRU classique ne s'applique plus dans le cadre des conteneurs, ceci pour deux raisons. Tout d'abord, la distribution des pages d'un conteneur sur différents nœuds implique la distribution de l'information sur l'âge des pages. Se pose alors le problème de la détermination de la page la plus ancienne d'un conteneur. D'autre part, plusieurs copies d'une même page peuvent exister dans la mémoire de la grappe.

Sur une machine classique équipée d'un système d'exploitation centralisée, le remplacement de page consiste à choisir une page et à la stocker temporairement sur disque. Dans le cadre des conteneurs, la modification de la hiérarchie mémoire introduit de nouvelles solutions pour le traitement d'une page à remplacer. Nous présentons ces solutions dans la suite de ce paragraphe et évaluons leur coût afin d'établir les critères de choix pour le remplacement d'une page.

8.5.1.1 Remplacement de pages en conteneurs

La politique de remplacement de page utilisée pour les conteneurs est une version modifiée de la politique proposée par Lahjomri pour la MPRL Koan [61], réalisée sur la machine parallèle iPSC/2. Cette machine ne disposant pas de disques, le remplacement de page est réalisé uniquement grâce à un mécanisme d'injection. Dans ce cas, l'injection n'est

possible que si au moins un nœud dispose de cadres de pages libres. Dans le cas contraire le remplacement est impossible, ce qui provoque l'arrêt du système.

Sur une grappe, chaque nœud dispose de périphériques de stockage secondaire qui permettent de stocker des pages évincées de la mémoire de la grappe. Dans le cas des conteneurs, nous distinguons deux cas de figure :

- **L'éviction d'une page de la mémoire d'un nœud** : après l'éviction, il subsiste au moins une copie de la page dans la mémoire de la grappe.
- **L'éviction d'une page de la mémoire de la grappe** : après l'éviction, il n'existe plus aucune copie de la page dans la mémoire de la grappe. Un accès ultérieur à cette page nécessite l'accès à un périphérique d'entrée/sortie.

Le premier cas correspond à la politique utilisée dans Koan. Elle est réalisable uniquement lorsqu'il existe des cadres de pages libres dans la grappe. Ce mécanisme est assuré uniquement par les conteneurs, sans intervention des lieurs.

Lorsqu'aucun cadre de page n'est disponible dans la grappe pour héberger une page à évincer, nous utilisons la deuxième solution. Dans ce cas, le mécanisme de conteneur ne peut réaliser l'éviction. Il fait alors appel au lieur d'entrée/sortie associé afin de "sortir" la page du conteneur et donc de la mémoire de la grappe.

Lorsqu'une page est sélectionnée pour être remplacée, plusieurs cas de figure peuvent se présenter suivant l'état de la page et de la quantité de mémoire disponible sur la grappe :

- **Éviction d'un duplicata** : la page à évincer est un duplicata, elle peut être simplement détruite. Ceci permet de libérer un cadre de page sans accès disque, ni transfert réseau.
- **Migration de propriété** : la page à évincer est la copie maître et il existe des duplicatas. L'éviction de cette copie nécessite de migrer la propriété de la page vers un autre nœud.
- **Injection de page** : la page à évincer est l'unique copie existant dans la grappe. L'éviction de cette page nécessite de l'injecter dans la mémoire d'un nœud distant disposant de suffisamment d'espace en mémoire centrale.
- **Remplacement sur lieur d'E/S** : la page à évincer est l'unique copie existante dans la grappe et il n'y a plus de cadre de page libre dans la grappe. La page doit être supprimée de la mémoire grâce à la fonction *flush_page* d'un lieur d'entrée/sortie.

8.5.1.2 Coût d'un remplacement

L'objectif de notre algorithme de remplacement est de minimiser le coût d'un remplacement. Ce coût peut être décomposé en deux parties : (1) le coût de remplacement et (2) le coût d'un accès ultérieur à une page remplacée.

Le coût de remplacement représente le temps nécessaire pour libérer effectivement un cadre de page. Ce temps varie en fonction de l'action à entreprendre pour remplacer la page. Le coût de chaque type de remplacement est le suivant :

- **Éviction d'un duplicata** : Libération d'un cadre de page.
- **Migration de propriété** : Envoi d'un message sur le réseau.
- **Injection de page** : Envoi d'une page mémoire sur le réseau.

- **Remplacement sur lieur d'E/S** : Le coût de cette opération dépend du type de lieur associé. A ce coût peut s'ajouter le coût de l'envoi de la page sur le réseau si le conteneur associé est attaché à un nœud distant.

Le coût d'un accès à une page ayant été remplacée représente le temps nécessaire pour obtenir une copie de la page dans la mémoire d'un nœud, après que celle-ci ait été remplacée. Ce temps varie également suivant le type de remplacement effectué. Le coût d'un accès à une page qui a été évincée suivant le type d'éviction est le suivant :

- **Éviction d'un duplicata** : le coût est nul sur un nœud disposant d'une copie. Sur les autres nœuds, coût d'un appel à la fonction *get_page*, c'est-à-dire le transfert d'une page sur le réseau.
- **Migration de propriété** : le coût est nul sur un nœud disposant d'une copie. Sur les autres nœuds, coût d'un appel à la fonction *get_page*, c'est-à-dire le transfert d'une page sur le réseau.
- **Injection de page** : le coût est nul sur le nœud disposant de la page. Sur les autres nœuds, coût d'un appel à la fonction *get_page*, c'est-à-dire le transfert d'une page sur le réseau.
- **Remplacement sur lieur d'E/S** : Coût d'un appel à la fonction *first_touch* du lieur d'entrée/sortie associé. Ce coût varie suivant le type de lieur. Il consiste généralement en une opération d'entrée/sortie et à un transfert de la page sur le réseau.

L'algorithme de remplacement choisit en priorité les pages pouvant être remplacées au plus faible coût, afin de libérer des cadres de page le plus rapidement possible et de limiter l'impact du remplacement sur le fonctionnement ultérieur du système (c'est-à-dire limiter le coût des accès à une page remplacée).

8.5.1.3 Remplacement sans intervention des lieurs

Dans le cas où il existe des cadres de pages libres dans la grappe, le remplacement de page peut être effectué sans l'intervention des lieurs d'entrée/sortie. Dans ce cas, les pages sont évincées de la mémoire des nœuds mais pas de la mémoire de la grappe, limitant ainsi le coût d'un accès à une page remplacée. Puisqu'il existe toujours au moins une copie de la page dans la grappe, il est possible d'obtenir une copie sur n'importe quel nœud grâce à un simple transfert réseau.

Le tableau 8.1 présente la priorité de remplacement des pages en fonction de leur état et donc de la fonction de remplacement à utiliser. Les duplicatas dans l'état lecture sont remplacés en priorité. Le coût de remplacement de ces pages est négligeable, ce qui permet un remplacement très rapide. Ensuite, sont remplacées les copies maîtres dans l'état lecture et dont il existe des duplicatas. Les pages en copie unique sont remplacées en dernier. Pour chaque catégorie, un algorithme LRU est utilisé pour choisir une page.

8.5.1.4 Remplacement avec intervention des lieurs

Lorsqu'il n'y a plus aucun cadre de page libre dans la grappe, le mécanisme de remplacement ne peut plus injecter de pages en mémoire distante. Le remplacement de la dernière

	État	Copie unique	Copie Maître	Action
1	Lecture	Non	Non	Évincer la page
2	Lecture	Non	Oui	Migrer la propriété
3	-	Oui	Oui	Injecter la page

TAB. 8.1 – Priorité de remplacement des pages avec possibilité d'injection

copie d'une page nécessite alors l'intervention d'un lieur d'entrée/sortie.

Le tableau 8.2 présente la priorité de remplacement des pages en fonction de leur état. Comme dans le cas précédent, les duplicatas dans l'état lecture sont remplacées en priorité. Les pages en copie unique sont remplacées en dernier et dans l'ordre suivant :

- pages en lecture non modifiées : ces pages n'ont pas été modifiées depuis leur chargement en conteneur. Elles peuvent donc être détruites puisque la copie présente sur le périphérique attaché est à jour ;
- pages modifiées non attachées à un nœud distant : ces pages ayant été modifiées depuis leur chargement en mémoire, le périphérique lié doit être mis à jour. Cette mise à jour peut être effectuée localement ;
- pages modifiées attachées à un nœud distant : la mise à jour du périphérique nécessite l'envoi de la page sur le nœud attaché et la mise à jour du périphérique attaché.

	État	Copie unique	copie maître	Modifié	Attachement distant	Action
1	Lecture	Non	Non	-	-	Évincer la page
2	Lecture	Non	Oui	-	-	Migrer la propriété
3	Lecture	Oui	Oui	Non	-	Évincer la page
4	-	Oui	Oui	Oui	Non	Éviction locale sur lieur
5	-	Oui	Oui	Oui	Oui	Éviction distante sur lieur

TAB. 8.2 – Priorité de remplacement des pages sans possibilité d'injection

Il est à noter que pour permettre l'envoi d'une page vers le nœud attaché lors d'un remplacement via un lieur d'entrée/sortie, il est nécessaire de disposer à tout instant de quelques cadres de page libres sur tous les nœuds. Ces cadres permettent de stocker les pages provenant des nœuds à l'origine d'un remplacement jusqu'à leur remplacement effectif par un lieur d'entrée/sortie.

8.5.2 Choix du nœud d'injection

Lorsque le mécanisme de remplacement décide d'injecter une page, il doit choisir un nœud cible pour héberger la page. Le choix du nœud dépend de plusieurs critères dont le principal est bien entendu la quantité de mémoire disponible sur ce nœud. D'autres critères

comme la localisation des processus pouvant référencer une page ne sont pas traités dans ce document.

8.5.2.1 Détermination de l'espace libre sur les nœuds

Nous divisons l'espace disponible sur un nœud en deux parties : (1) l'ensemble des cadres de pages non alloués et (2) l'ensemble des cadres de pages hébergeant des duplicatas. Cette deuxième catégorie de cadre de pages représente des cadres qui sont alloués, mais qui peuvent être évincés rapidement et en causant peu de perturbation dans le système. En supprimant un duplicata pour y placer une page injectée, on peut éviter d'évincer d'un conteneur la dernière copie d'une page, limitant les entrée/sorties pour le remplacement et lors d'un éventuel futur accès à la page remplacée.

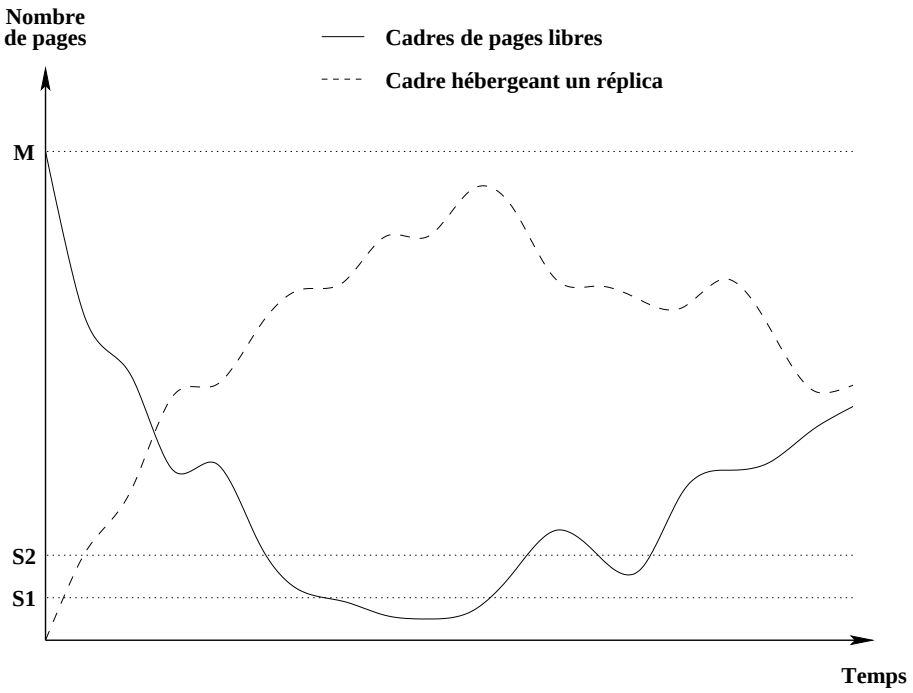

FIG. 8.4 – *Évolution du nombre de cadres de pages libres et du nombre de cadres de page hébergeant des duplicatas en fonction du temps.*

Sur chaque nœud, deux compteurs sont utilisés pour mesurer l'évolution du nombre de cadres de page libres (*nb_free_page*) et le nombre de cadres de page hébergeant un duplicata (*nb_copy_page*). Sur chaque nœud, lorsque *nb_free_page* passe au dessus d'un seuil *S2*, un message est diffusé aux autres nœuds de la grappe pour indiquer qu'il dispose de cadres de page libres. Lorsque ce compteur passe en dessous d'un seuil *S1*, un message est diffusé pour indiquer que la mémoire du nœud est saturée (voir figure 8.4).

De manière similaire, lorsque le compteur *nb_copy_page* passe au dessus du seuil *S2*, un message est diffusé pour indiquer que le nœud dispose de cadres hébergeant des dupli-

catas pouvant être évincés. Lorsque ce compteur passe en dessous du seuil *S1*, un message est diffusé pour indiquer que le nœud ne dispose plus de cadres de page hébergeant des duplicatas.

8.5.2.2 Choix du nœud d'injection

Chaque nœud dispose d'un tableau indiquant pour chaque nœud de la grappe, l'état de sa mémoire, à savoir :

- **Libre** : le nœud dispose de cadre de page libres.
- **Saturé** : le nœud ne dispose pas de cadres de page libres, mais dispose de cadres de page hébergeant des duplicatas.
- **Plein** : le nœud ne peut héberger aucune page supplémentaire.

S'il existe au moins un nœud dans l'état *libre* ou *saturé*, l'algorithme de remplacement présenté en section 8.5.1.3 est utilisé. Les nœuds dans l'état *libre* sont choisis en priorité. S'il n'y a aucun nœud dans l'état *libre*, un nœud dans l'état *saturé* est choisi.

Si tous les éléments du tableau contiennent l'état *plein*, cela signifie qu'il n'y a plus de cadres de pages disponibles dans la grappe. L'injection est alors impossible. Le système utilise alors l'algorithme de remplacement présenté en section 8.5.1.4.

8.5.3 Mécanismes de remplacement de pages

Lorsque la page à remplacer et le nœud d'injection ont été choisis, le contenu de la page doit être sauvegardé avant de libérer le cadre de page associé. Nous détaillons dans ce paragraphe les mécanismes utilisés pour remplacer la page choisie.

8.5.3.1 Éviction d'un duplicata

L'éviction d'un duplicata consiste simplement à libérer le cadre de page associé et à informer le propriétaire de la page que le nœud local ne dispose plus du duplicata. Pour cela, une requête est envoyée au gestionnaire de la page **[1]**, qui la transmet au propriétaire **[2]** (voir figure 8.5). Sur réception du message, le propriétaire supprime le nœud émetteur du *copyset* **[3]**. L'éviction d'un duplicata ne nécessite pas de message d'acquittement. Dans ce cas, l'éviction est très rapide puisque qu'elle consiste simplement en la libération d'un cadre de page et en l'émission d'un message. L'algorithme de cette fonction est présenté en annexe C.1.

8.5.3.2 Migration de propriété

La migration de la propriété d'une page nécessite le transfert des informations gérées par le propriétaire de la page (c'est-à-dire le *copyset*) sur un nœud disposant d'une copie. Pour cela, le nœud réalisant le transfert de propriété choisit un nœud dans le *copyset* et lui envoie les informations de propriété **[1]** (voir figure 8.6). Lorsque le nœud choisi reçoit ces informations, il vérifie qu'il dispose toujours d'une copie de la page. Si c'est le cas, il intègre les informations de propriété et envoie un acquittement au gestionnaire de la page

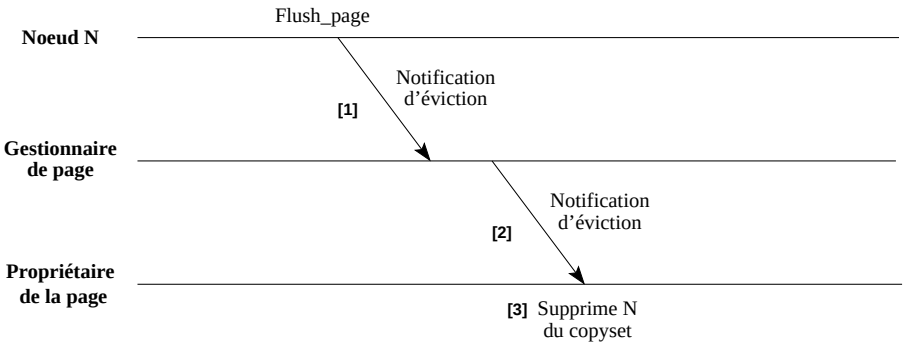

FIG. 8.5 – *Remplacement de page : cas de l'éviction d'un duplicata*

[2]. Le gestionnaire note le changement de propriété et envoie un acquittement à l'ancien propriétaire **[3]**. Celui-ci libère alors le cadre de page **[4]**.

Si le nœud choisi pour devenir le nouveau propriétaire ne dispose plus d'une copie de la page, il envoie un acquittement négatif au propriétaire actuel. Celui-ci choisit alors un autre nœud pour devenir le nouveau propriétaire. S'il n'existe plus aucun nœud disposant d'une copie, le propriétaire appelle la fonction d'injection de page.

L'algorithme de cette fonction est présenté en annexe C.2.

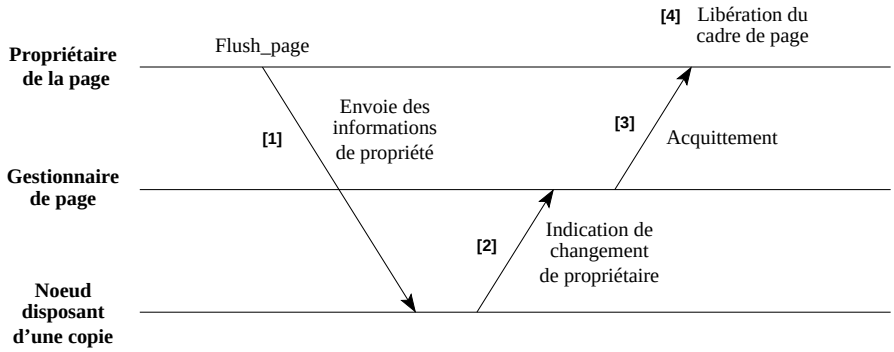

FIG. 8.6 – *Remplacement de page : cas de la migration de propriété*

8.5.3.3 Injection d'une page

L'injection d'une page nécessite d'envoyer le contenu de la page vers un nœud sélectionné pour devenir le nouveau propriétaire. Pour cela, le nœud réalisant l'injection choisit

un nœud parmi les nœuds disposant de cadres de pages libres et lui envoie une requête d'injection **[1]** (voir figure 8.7). Ce nœud alloue un cadre de page, y place la page reçue **[2]** et envoie un acquittement au gestionnaire de la page **[3]**. Le gestionnaire note le changement de propriété et envoie un acquittement à l'ancien propriétaire **[4]**. Celui-ci libère alors le cadre de page **[5]**. L'algorithme de cette fonction est présenté en annexe C.3.

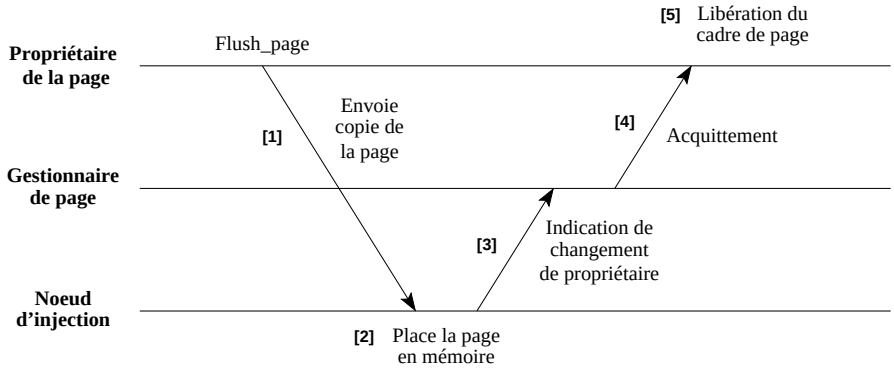

FIG. 8.7 – *Remplacement de page : cas de l'injection d'une page*

8.5.3.4 Remplacement sur lieur d'E/S

Le remplacement de page via un lieur est réalisé lorsqu'aucun cadre de page n'est disponible sur la grappe. Dans ce cas, la page est supprimée du conteneur, c'est-à-dire que la dernière copie existante est évincée de la mémoire de la grappe. Pour cela, si le conteneur est attaché à un nœud particulier, la page est envoyée à ce nœud pour être évincée **[1]** (voir figure 8.8). Le nœud attaché appelle alors la fonction *flush_page* du lieur d'entrée/sortie **[2]** et envoie un acquittement au gestionnaire de la page **[3]**. Celui-ci note que la page n'a plus de propriétaire et acquitte le nœud à l'origine du remplacement **[4]**, qui peut alors libérer le cadre de page associé **[5]**. Si le conteneur n'est pas attaché, le nœud à l'origine du remplacement appelle localement la fonction *flush_page* du lieur d'entrée/sortie et libère le cadre de page associé. Ce nœud est toujours considéré comme étant le propriétaire de la page, ce qui permet de retrouver le nœud ayant réalisé l'éviction lors d'un futur accès à la page. L'algorithme de cette fonction est présenté en annexe C.4.

8.5.3.5 Problème de la mise à jour de la liste des copies

Lorsqu'un duplicata est évincé de la mémoire de la grappe, le *copyset* doit être mis à jour sur le nœud propriétaire de la page. Pour ce faire, un message est envoyé au propriétaire à chaque éviction. Ceci implique une incohérence temporaire de la liste des copies chez le

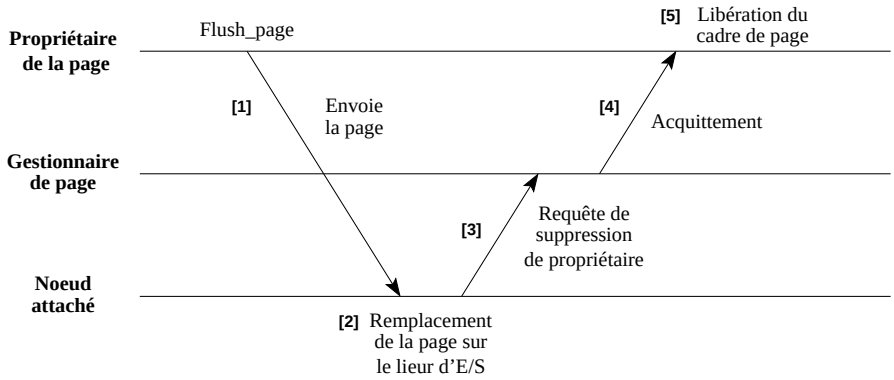

FIG. 8.8 – *Remplacement de page : cas du remplacement sur lieur d'E/S*

propriétaire. Entre l'émission du message par le nœud évinçant le duplicata et la réception du message par le propriétaire, la liste des copies est incohérente. En effet, le nœud ayant évincé le duplicata fait toujours partie du *copyset* alors que le duplicata a été évincée.

Cette incohérence temporaire pourrait être évitée en forçant le nœud évinçant un duplicata à attendre un acquittement du propriétaire avant de libérer le cadre de page. Or ceci ralentirait considérablement le mécanisme de remplacement, ce qui n'est pas souhaitable. De plus, cette incohérence est de très courte durée (le temps de transmission d'un message sur le réseau) et n'est problématique que si le propriétaire utilise simultanément les informations du *copyset*, c'est-à-dire lors d'une invalidation ou lors d'un changement de propriété. La probabilité que le *copyset* soit incohérent durant l'une de ces opérations est très faible. Il est donc raisonnable de tolérer cette incohérence afin d'améliorer les performances du remplacement. Cependant, toute action entreprise sur le nœud propriétaire et faisant intervenir le *copyset* doit tenir compte de cette incohérence potentielle. Cela est notamment le cas lors de l'invalidation des duplicatas et de la migration de propriété :

- **Invalidation des duplicatas**. Il est à noter que dans tous les cas, le *copyset* dispose d'un sur-ensemble des nœuds hébergeant un duplicata. Ainsi, lorsqu'un nœud propriétaire demande l'invalidation des duplicatas, toutes les copies sont bien invalidées, même si le *copyset* est incohérent. Dans ce cas, un message est envoyé inutilement au nœud ayant évincé un duplicata. Cette situation étant exceptionnelle, la perte de performance induite par l'envoi de messages inutiles devrait être imperceptible.
- **Migration de propriété**. Lors de la migration de propriété, l'incohérence potentielle du *copyset* est beaucoup plus problématique. Dans ce cas, le propriétaire choisit dans le *copyset* un nœud qui deviendra le nouveau propriétaire. Si le nœud choisi a évincé sa copie entre-temps, un autre nœud doit être choisi.

8.6 Résumé

Nous avons présenté dans ce chapitre la conception d'un mécanisme logiciel permettant de stocker et de partager des données entre des noyaux d'un système d'exploitation hôte s'exécutant sur différents nœuds d'une grappe de calculateurs. Nous appelons ce mécanisme **conteneur**. Un conteneur est structuré en pages mémoires, qui peuvent être déplacées ou dupliquées de la mémoire d'un nœud vers la mémoire d'un autre nœud. Un conteneur peut être attaché à un nœud particulier de la grappe si les données qu'il héberge sont stockées sur un périphérique de ce nœud.

On peut accéder à un conteneur grâce à un ensemble de fonctions d'interface permettant de s'assurer de la présence en mémoire locale d'une page particulière d'un conteneur. On distingue principalement deux types d'accès aux conteneurs : les accès en lecture et les accès en écriture. Lors d'un accès en lecture grâce à la fonction *get_page*, la page considérée est recopiée depuis la mémoire d'un nœud distant dans la mémoire du nœud demandant l'accès. Cette page ne peut alors être accédée qu'en lecture sur toute la grappe. Lors d'un accès en écriture grâce à la fonction *grab_page*, la page est déplacée du nœud propriétaire vers le nœud demandant l'accès et toutes les duplicatas sont invalidées. Il n'existe plus alors qu'une seule copie pouvant être accédée en lecture et en écriture.

Lorsque la mémoire physique d'un nœud est saturée, des cadres de page peuvent être libérées grâce à la fonction *flush_page*. Celle-ci tente tout d'abord de supprimer les duplicatas, puis injecte les pages en copies unique sur d'autres nœuds. Enfin, lorsqu'il n'y a plus aucun cadre de page libre dans la grappe, les pages sont remplacées sur disque.

Les conteneurs sont intégrés au sein du système d'exploitation hôte grâce à un ensemble de **lieurs**. Les lieurs d'interface permettent de modifier l'interface d'accès aux conteneurs, alors que les lieurs d'entrée/sortie sont utilisés pour charger des données en conteneur depuis le périphérique associé.

9 SERVICES SYSTÈMES DISTRIBUÉS À BASE DE CONTENEURS

Nous avons présenté dans le chapitre précédent la conception d'un mécanisme logiciel appelé conteneur, permettant de stocker et de partager des données entre des noyaux d'un système d'exploitation hôte s'exécutant sur différents nœuds d'une grappe de calculateurs. Les conteneurs sont intégrés au sein du système d'exploitation hôte grâce à un ensemble de **lieurs**. Les lieurs d'interface permettent de modifier l'interface d'accès aux conteneurs, alors que les lieurs d'entrée/sortie sont utilisés pour charger des données en conteneur depuis le périphérique associé.

Dans ce chapitre, nous présentons en premier lieu la conception des lieurs d'interface offrant des mécanismes de projection de conteneur en mémoire ou l'accès aux conteneurs grâce à une interface de type fichier. Nous présentons ensuite la conception des lieurs d'entrées-sortie en mémoire et sur fichier.

Dans une troisième partie, nous présentons comment l'utilisation conjointe des conteneurs et des lieurs permet de mettre en œuvre des services systèmes tels qu'une mémoire virtuelle partagée, un mécanisme de projection de fichiers en mémoire, un cache de fichiers coopératif et un système de gestion de fichiers distribué.

9.1 Conception des lieurs d'interface

Le rôle des lieurs d'interface est de permettre de changer l'interface d'un conteneur et de l'adapter aux exigences des services systèmes de haut niveau. Cette interface doit donner l'illusion au noyau qu'il communique avec les gestionnaires de périphériques avec lesquels il interagit traditionnellement. Nous détaillons dans cette partie la conception de deux types de lieurs d'interface : un lieur offrant une interface d'accès aux conteneurs de type **projection** en mémoire virtuelle et une interface de type **fichier**. Ces deux interfaces sont les plus couramment utilisées par un système d'exploitation.

9.1.1 Projection en mémoire virtuelle

L'une des interfaces d'accès aux périphériques les plus utilisées par le noyau d'un système d'exploitation est la projection en mémoire virtuelle. Lorsque l'on observe l'image mémoire d'un processus, on constate qu'elle est découpée en segments de mémoire virtuelle. Chaque segment correspond en réalité à un périphérique projeté dans l'espace d'adressage

du processus. Le segment de texte correspond à la projection du fichier exécutable représentant le processus, le segment de données correspond à une zone de mémoire physique projetée, etc. La projection est assurée grâce au gestionnaire de mémoire virtuelle, qui gère les évènements provenant de la MMU lors de violations de droits d'accès (voir figure 9.1). On distingue deux types de projection : (1) la projection partagée et (2) la projection privée.

- **Projection partagée** : toute modification apportée sur un segment de projection est immédiatement visible par tous les processus partageant le périphérique à travers une projection partagée. Si le segment est associé à un périphérique de stockage, les modifications apportées à ce segment sont reportées sur le périphérique.
- **Projection privée** : toutes les modifications d'un segment de projection privées sont locales au processus. Cependant les modifications sont visibles par tous les *threads* du processus. Si le segment est associé à un périphérique de stockage, les modifications apportées à ce segment ne sont pas reportées sur le périphérique.

Dans un noyau, l'interface de projection consiste généralement en deux fonctions : (1) *first_ touch*, qui permet de créer une page lors d'un premier accès et (2) *copy_ on_ write*, qui permet de réaliser une copie sur écriture. La fonction *copy_ on_ write* est utilisée pour optimiser la création de processus. Lorsqu'un nouveau processus est créé suite à un « *fork* », aucune donnée de son espace d'adressage n'est recopiée. Les données du processus initial sont partagées grâce au mécanisme de mémoire virtuelle jusqu'à ce que l'un des processus modifie une donnée. Dans ce cas, une copie de la page touchée est réalisée afin de permettre aux données des deux processus d'évoluer indépendemment.

Le gestionnaire de mémoire physique assure la gestion des pages physique à travers deux fonctions : (1) *alloc_ page* qui permet d'allouer un cadre de page et (2) *release_ page* qui permet de libérer un cadre de page.

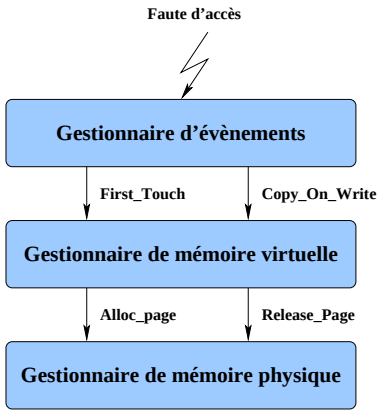

FIG. 9.1 – *Système de gestion de mémoire virtuelle*

Le lieur d'interface de projection en mémoire doit offrir ces deux fonctions comme interface d'accès aux conteneurs, tout en assurant le respect du protocole de cohérence

(voir figure 9.2). Nous présentons dans la suite de ce paragraphe la conception des fonctions *First_ touch* et *Copy_ On_ Write* du lieur d'interface.

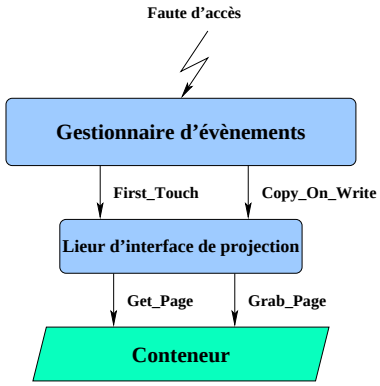

FIG. 9.2 – *Lieur d'interface de projection*

9.1.1.1 Détournement de la fonction First_touch

La fonction *first_ touch* est appelée par le système d'exploitation lors d'un accès à une page de la mémoire virtuelle qui n'est associée à aucun cadre de page en mémoire physique. Cette situation peut se présenter dans deux cas : (1) lors du premier accès à une page virtuelle ou (2) lors de l'accès à une page qui a été évincée de la mémoire. Dans le premier cas, le système d'exploitation alloue un nouveau cadre de page et l'associe à la page en défaut. Dans le second cas, le système localise la page dans la zone d'échange du disque local, charge le contenu de la page dans un nouveau cadre de page et l'associe à la page en défaut.

Lorsqu'un segment de mémoire virtuelle est associé à un conteneur, un troisième cas peut produire un appel à la fonction *first_ touch* du système hôte : (3) l'accès à une page qui a été invalidée par le protocole de cohérence.

Lorsqu'un segment de mémoire virtuelle est associé à un conteneur, la fonction *first_ touch* du système hôte associée à cette zone est remplacée par la fonction du lieur d'interface. Si le défaut de page a eu lieu pour l'une des trois raisons présentées plus haut, une copie de la page en défaut sera placée en mémoire locale par l'intermédiaire des conteneurs. En fonction du type d'accès ayant causé le défaut de page, la fonction *first_ touch* du lieur de projection réalise l'une les actions suivantes (voir algorithme 1) :

- **Accès en écriture** : le lieur appelle la fonction *grap_ page* afin d'obtenir une copie en écriture de la page. L'adresse du cadre de page correspondant est retourné.
- **Accès en lecture** : le lieur vérifie si la page est présente localement grâce à la fonction *find_ page*. Si la page est présente, son adresse est retournée. Dans le cas contraire, un appel à la fonction *get_ page* est effectué. Enfin, l'adresse du cadre de page correspondant est retourné.

Les cas (1) et (2) gérés par la fonction *first_ touch* du système hôte sont gérés par les mécanismes internes des conteneurs.

Algorithme 1 : Algorithme du « *first touch* » du lieur d'interface de projection

```
Mapping Linker First Touch :
{
    si accès en écriture sur p alors
        page := grab_page (p) ;
    sinon
        page := find_page (p) ;
        si page = NULL alors
            page := get_page (p) ;
        fin si
    fin si
    Retourner page ;
}
```

9.1.1.2 Détournement de la fonction Copy_On_Write

La fonction *copy_ on_ write* est appelée par le système d'exploitation lors d'un accès en écriture à une page partagée en lecture entre deux processus suite à un « *fork* » par exemple. Le système réalise alors une copie du cadre de page partagé et associe le nouveau cadre de page à la page en défaut.

Lorsqu'un segment de mémoire virtuelle est associé à un conteneur, un nouveau cas peut produire un appel à la fonction *copy_ on_ write* : l'accès en écriture à une page qui a été placée dans l'état *lecture seule* par le protocole de cohérence.

Pour tout segment de mémoire virtuelle lié à un conteneur, la fonction *copy_ on_ write* du système hôte associée à cette zone est remplacée par la fonction du lieur d'interface. Celui-ci se charge alors d'obtenir une copie en écriture de la page en faisant appel à la fonction *grap_ page* (voir algorithme 2). Le cas de la copie sur écriture tel qu'il était géré par la fonction *copy_ on_ write* du système hôte est gérés par les mécanismes internes des conteneurs.

Algorithme 2 : Algorithme du « *copy on write* » du lieur d'interface de projection

```
Mapping Linker Copy On Write :
{
    page := grab_page (p) ;
    Retourner page ;
}
```

9.1.1.3 Notification de changement d'état

Le protocole de gestion de la cohérence nécessite de pouvoir modifier l'état des pages sur les différents nœuds de la grappe. Cette modification d'état peut avoir des répercussions sur l'interface d'accès aux conteneurs, notamment dans le cas d'une projection en mémoire virtuelle. L'état d'une page dans un conteneur doit être répercuté sur l'état de la page dans l'espace d'adressage des processus afin d'en contrôler les droits d'accès. Ceci est réalisé grâce à la fonction *change_ access* du lieur d'interface.

Dans le cas d'un lieur d'interface de projection, la fonction *change_ access* modifie les droits d'accès liés aux pages d'un segment de mémoire virtuelle lié à un conteneur. Lorsqu'une page d'un conteneur est en lecture seule, la page virtuelle correspondante doit être accessible uniquement en lecture (voir algorithme 3). Lorsqu'une page d'un conteneur est invalide, la page virtuelle correspondante ne doit pas être accessible. Cette modification de droit d'accès est réalisée en modifiant les droits d'accès aux pages au sein de l'unité de gestion de mémoire virtuelle (MMU).

Algorithme 3 : Algorithme réalisant le changement des droits d'accès sur une page d'un conteneur

```
Mapping Linker Change Access :
{
    selon Nouveau droit d'accès
      cas Lecture/Écriture :
          Place les droits d'accès de la page virtuelle à lecture et écriture
      cas Lecture seule :
          Place les droits d'accès de la page virtuelle à lecture seule
      cas Invalide :
          Supprime le lien entre la page et le cadre de page
    fin selon
}
```

9.1.2 Interface de fichier

La seconde interface utilisée par le noyau pour accéder aux périphériques est l'interface de fichier. Grâce à cette interface, le noyau peut accéder à tous les périphériques de la même manière. Les principales fonctions de cette interface sont les fonctions d'ouverture, de fermeture, de lecture et d'écriture.

Les systèmes d'exploitation de type Unix™ utilisent généralement une architecture à deux niveaux, composée d'un système de fichier virtuel ou « *Virtual File System* » (VFS) et un ensemble de systèmes de fichiers (voir figure 9.3). Un cache de fichier unique permet au VFS de garder en mémoire les pages chargées par les différents systèmes de fichiers.

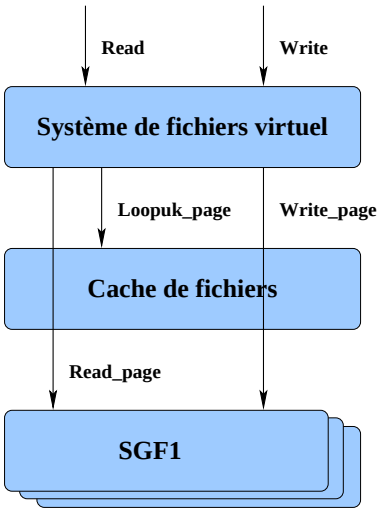

FIG. 9.3 – *Architecture d'un système de gestion de fichier*

9.1.2.1 Fonctionnement d'un accès en lecture

Lors d'un accès en lecture sur un fichier, le VFS traduit les accès de type *lecture d'un tampon* en accès de type *lecture de pages*. Tous les accès aux fichiers passent d'abord par le cache grâce à une fonction *lookup_page* permettant de vérifier la présence d'une page dans le cache. Si le cache ne peut satisfaire la demande, le VFS demande une lecture de page au système de fichier associé grâce à une fonction *read_page* (voir figure 9.4). Cette opération est répétée jusqu'à ce que toutes les données à lire aient été chargées dans le cache et recopiées du cache vers le tampon de l'utilisateur.

9.1.2.2 Fonctionnement d'un accès en écriture

Lors d'un accès en écriture sur un fichier, le VFS traduit les accès d'*écriture d'un tampon* en accès de type *écriture de pages*. Toutes les écritures sont réalisées dans le cache avant d'être répercutées sur le disque. Pour cela, le VFS vérifie la présence de la page à écrire dans le cache grâce à une fonction *lookup_page*. Si la page n'est pas dans le cache, une nouvelle entrée est créée afin de recevoir les données à écrire. Les données sont alors recopiées du tampon utilisateur vers l'entrée allouée dans le cache. Cette opération est répétée jusqu'à écriture complète du tampon dans le cache. (voir figure 9.5). Périodiquement, les données modifiées du cache sont mises à jour sur disque grâce aux fonctions *write_page* des systèmes de fichiers.

9.1.2.3 Conception du lieur d'interface fichier

Les fonctions de lecture et écriture sur fichier passent systématiquement par la fonction *lookup_page* du cache pour lire ou écrire une page d'un fichier. Cette fonction semble donc

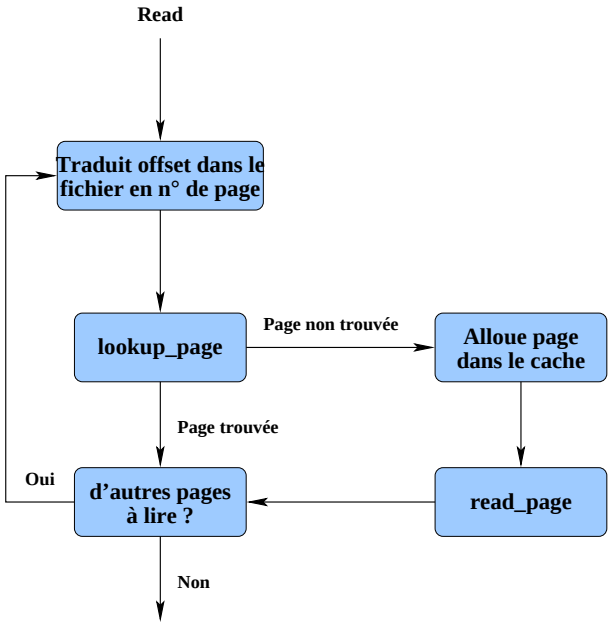

FIG. 9.4 – *Diagramme de lecture dans un fichier*

le point idéal pour insérer un lieur d'interface (voir figure 9.6). En effet, en détournant tous les accès à cette fonction, il est possible de détourner tous les accès aux fichiers vers les conteneurs. Si la fonction du lieur d'interface assure de toujours retourner un cadre de page, la fonction *read_page* ne sera jamais appelée par le VFS. Ainsi, il n'est pas nécessaire de détourner cette fonction dans le lieur d'interface.

Cependant, la fonction *lookup_page* utilisée dans les systèmes d'exploitation ne distingue pas les accès en lecture des accès en écriture. Or, cette information est nécessaire pour le maintien de la cohérence. Le noyau doit donc être légèrement modifié afin d'introduire une information sur le type d'accès réalisé lors de l'appel à la fonction *lookup_page*.

Enfin, la fonction *write_page* utilisée par le VFS pour mettre à jour les unités de stockages est détournée afin de réaliser la sauvegarde des pages du cache sur les nœuds hébergeant les périphériques qui leurs sont associés.

9.1.2.4 Détournement de la fonction *Lookup_Page*

Lorsqu'un fichier est associé à un conteneur, un appel à la fonction *lookup_page* place dans un cadre de page en mémoire locale les données correspondant à la page recherchée et provenant du conteneur associé au fichier. Pour cela, la fonction *lookup_page* détermine à quel conteneur est associé le fichier, et réalise l'une les actions suivantes en fonction du type d'accès (voir algorithme 4) :

- **Accès en écriture** : le lieur appelle la fonction *grap_page* afin d'obtenir une copie en écriture de la page.

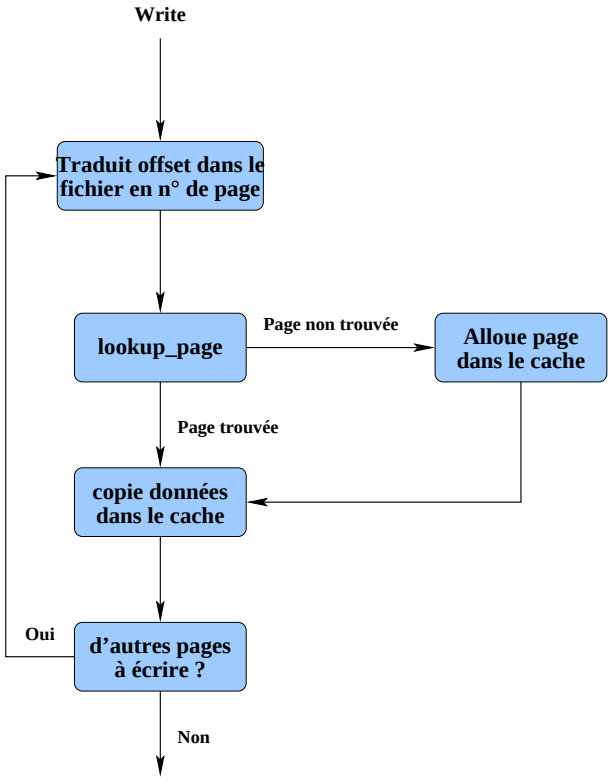

FIG. 9.5 – *Diagramme d'écriture dans un fichier*

– **Accès en lecture** : le lieur vérifie si la page est présente localement grâce à la fonction *find_page*. Si la page est présente, son adresse est retournée. Dans le cas contraire, un appel à la fonction *get_page* est effectué.

9.1.2.5 Détournement de la fonction *Write_Page*

Périodiquement, le VFS parcourt l'ensemble des pages présentes dans le cache de fichiers à la recherche des pages nécessitant une mise à jour du périphérique de stockage associé. Pour chacune de ces pages, la fonction *write_page* est appelée afin de sauvegardé les données de ces pages sur le périphérique de stockage associé. Un appel à cette fonction ne modifie donc en rien le contenu ou l'état de la page au sens du protocole de cohérence.

Cependant, un conteneur fichier étant connu sur l'ensemble des nœuds de la grappe, chaque nœud est susceptible d'appeler cette fonction. Or, celle-ci n'a de sens que sur le nœud hébergeant le périphérique associé. Ainsi, le rôle du lieur d'interface est simplement de relayer l'appel à la fonction *write_page* du lieur d'interface vers la fonction *write_page* du SGF associé lorsque ceci a un sens, c'est-à-dire sur le nœud hébergeant le périphérique (voir algorithme 5).

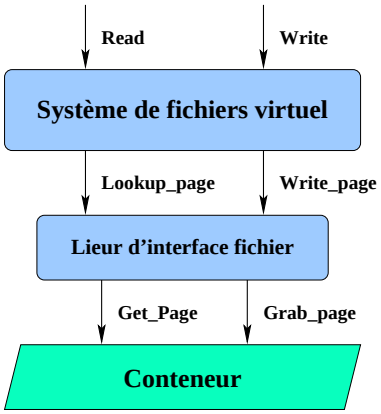

FIG. 9.6 – *Lieur d'interface de fichier*

Algorithme 4 : Algorithme réalisant la recherche d'une page dans le cache

```
Lookup_page :
{
    si accès en écriture sur p alors
        page := grab_page (p) ;
    sinon
        page := find_page (p) ;
        si page = NULL alors
            page := get_page (p) ;
        fin si
    fin si
    Retourner page ;
}
```

9.1.2.6 Notification de changement d'état

Contrairement à l'interface de projection où des accès implicites aux pages sont possibles grâce à des lectures et écritures en mémoire, le changement d'état d'une page d'un conteneur n'a pas de répercussion directe sur l'interface de fichier. Chaque accès à une page d'un fichier est réalisé explicitement grâce aux fonctions détournées par le lieur d'interface. Au sein de ces fonctions, les droits d'accès sont systématiquement vérifiés. La fonction de notification ne réalise donc aucune opération.

9.2 Conception des lieurs d'entrée/sortie

Le rôle des lieurs d'entrée/sortie est de permettre d'accéder aux ressources physiques gérées par un conteneur à travers un ensemble de fonctions génériques. Les lieurs d'en-

Algorithme 5 : Algorithme réalisant la mise à jour d'un périphérique de stockage

```
Write_page :
{
    si nœud attaché au conteneur hébergeant p = nœud local alors
        write_page (p) ;
    fin si
}
```

trée/sortie sont utilisés afin d'instancier un conteneur. Lorsqu'un conteneur est associé à un lieur d'entrée/sortie, il cesse d'être générique : une sémantique lui est associée. Nous détaillons dans cette partie la conception de deux types de lieurs d'entrée/sortie : un lieur permettant l'accès à la ressource mémoire physique et un lieur permettant l'accès aux unités de stockage de type bloc. Ces deux ressources sont les deux principales ressources gérées par un système d'exploitation.

9.2.1 Lieur d'entrée/sortie en mémoire

Le lieur d'entrée/sortie en mémoire a la charge de faire entrer et sortir d'un conteneur des pages de mémoire physique. Cette opération consiste en réalité à allouer et libérer des cadres de page comme le fait le noyau pour la gestion des segments de mémoire d'un processus (voir figure 9.7). Les fonctions du lieur d'entrée/sortie mémoire sont donc extrêmement simples à réaliser.

Un conteneur associé à un lieur d'entrée/sortie mémoire est instancié en**conteneur mémoire**. Il héberge des données stockées uniquement en mémoire, dont la durée de vie est associée à la durée de vie du ou des processus utilisant ces données.

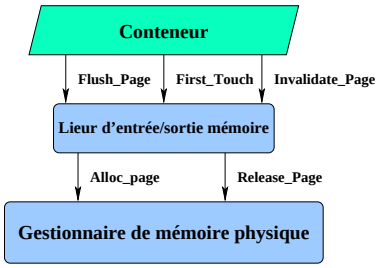

FIG. 9.7 – *Lieur d'entrée/sortie en mémoire*

9.2.1.1 La fonction first_touch

Un premier accès sur une page mémoire correspond à une allocation paresseuse de mémoire. Aucune donnée permanente n'étant stockée en mémoire, le premier accès corres-

pond simplement à une allocation de cadre de page, sans aucune opération supplémentaire d'entrée/sortie.

La fonction *first_ touch* alloue donc un cadre de page physique et retourne son adresse.

9.2.1.2 La fonction free_page

Lorsqu'une page mémoire est libérée, son contenu est perdu. Aucune opération d'entrée/sortie n'est réalisée pour sauvegarder son contenu. La fonction *free_ page* se contente donc de libérer le cadre de page associé à la page en conteneur.

9.2.1.3 La fonction flush_page

Lors d'un remplacement de page, le noyau sauvegarde le contenu de la page sur une unité de stockage secondaire et libère le cadre de page.

La fonction *flush_ page* du lieur d'entrée/sortie fait appel la fonction de remplacement sur disque du système hôte et libère le cadre de page.

9.2.1.4 La fonction invalidate_page

L'invalidation de page n'existe pas dans un noyau de système d'exploitation. Cette fonctionnalité a été ajoutée par le mécanisme de conteneur pour la gestion de la cohérence. Dans le cas d'une invalidation de page mémoire, la page est rendue inaccessible grâce au lieur d'interface. La page n'étant plus accessible, elle peut être libérée par la fonction *invalidate_ page* du lieur d'entrée/sortie.

9.2.2 Lieur d'entrée/sortie sur fichier

Le lieur d'entrée/sortie sur fichier a la charge de faire entrer et sortir d'un conteneur des pages provenant d'un fichier. Il s'agit principalement d'effectuer des accès aux périphériques de stockage hébergeant les fichiers associés aux conteneurs (voir figure 9.8). Ces fonctions sont donc légèrement plus complexes que celles associées au lieur d'entrée/sortie mémoire.

Un conteneur associé à un lieur d'entrée/sortie fichier est instancié en **conteneur fichier**. Il héberge des données provenant d'un périphérique de stockage secondaire, dont la durée de vie n'est pas limitée. Ces données peuvent rester en mémoire temps qu'il reste des cadres de page libres.

9.2.2.1 La fonction first_touch

Un premier accès sur une page d'un conteneur associé à un fichier correspond au chargement des données correspondantes depuis l'unité de stockage. Les données associées à cette page peuvent déjà être présentes en mémoire locale, au sein du cache de fichier. La fonction *first_ touch* commence donc par vérifier la présence de la page dans le cache de fichier. Si la page est présente, son adresse est retournée. Si la page n'est pas dans le cache, une nouvelle

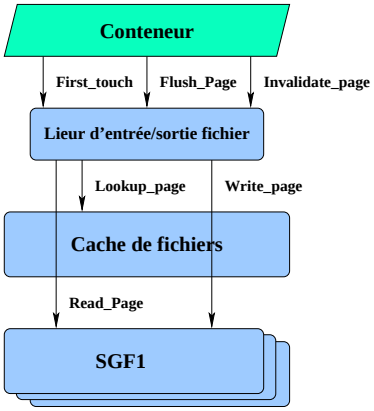

FIG. 9.8 – *Lieur d'entrée/sortie sur fichier*

entrée est allouée dans le cache et la page est chargée depuis l'unité de stockage. L'adresse du cadre de page alloué est ensuite retournée.

9.2.2.2 La fonction free_page

Lorsqu'une page d'un conteneur fichier est libérée, le périphérique associé doit être mis à jour si la page a été modifiée depuis son chargement. La fonction *free_page* met donc à jour la copie sur disque si nécessaire, supprime l'entrée dans le cache et libère le cadre de page associé.

9.2.2.3 La fonction flush_page

Le remplacement d'une page du cache de fichier ne nécessite pas son insertion dans la zone d'échange de l'unité de stockage secondaire. La page peut en effet être simplement écrite sur disque à son emplacement d'origine. Si le disque est à jour, aucun accès disque n'est nécessaire. Ainsi, la fonction *flush_page* réalise la même opération que la fonction *free_page*. Il s'agit en effet de libérer le cadre de page associé après avoir mis à jour le périphérique si nécessaire.

9.2.2.4 La fonction invalidate_page

L'invalidation d'une page d'un conteneur fichier permet de rendre la page inaccessible localement. La page est supprimée de la mémoire locale, mais il existe d'autres copies dans la grappe. Il n'est donc pas nécessaire de mettre à jour le périphérique associé. La fonction *invalidate_page* se contente ainsi de supprimer l'entrée correspondante dans le cache de fichier et de libérer le cadre de page associé.

9.2.3 Lieur d'entrée/sortie conteneur

Le lieur d'entrée/sortie en conteneur permet de charger dans un conteneur des données provenant d'un autre conteneur. Ceci est notamment utilisé lors de la projection privée d'un fichier en mémoire virtuelle ou de la copie paresseuse des données d'un conteneur vers un autre conteneur suite à un « *fork* ». Ce lieur est identique au lieur mémoire à l'exception de la fonction *first_ touch* qui permet de faire entrer des données en conteneur.

Lors de l'appel à la fonction *first_ touch* du lieur de conteneur, la page correspondante dans le conteneur lié doit être chargée dans le conteneur réalisant le *first_ touch*. Pour cela, la fonction *get_ page* du conteneur lié est appelée afin d'obtenir une copie en mémoire locale de la page désirée. L'adresse de cette page est alors retournée au conteneur appelant.

9.3 Conception de services systèmes fondés sur des conteneurs

Nous avons défini dans les paragraphes précédents la notion de conteneur et de lieur et décrit les mécanismes utilisés pour les réaliser. Nous disposons maintenant d'outils simples et efficaces permettant de concevoir de nombreux services systèmes distribués de haut niveau.

Nous présentons dans la suite de ce paragraphe comment utiliser les conteneurs et les différents lieurs introduits pour réaliser des services systèmes tels qu'une mémoire virtuelle partagée, un système de projection de fichiers distribuée, un cache de fichier coopératif et un système de gestion de fichier distribué. Enfin, nous détaillons comment tous ces services peuvent simplifier de manière significative les mécanismes utilisés par la migration de processus.

9.3.1 Mémoire virtuelle partagée

Une mémoire virtuelle partagée permet à plusieurs processus s'exécutant sur des nœuds différents de partager des variables en mémoire. Pour assurer ce service, il est nécessaire d'assurer trois propriétés : (1) le partage des données entre les nœuds, (2) la gestion de la cohérence de ces données et (3) un accès simple aux données grâce aux opérations de lecture et écriture du processeur. Les deux premières propriétés sont assurées par le service de conteneur et la troisième propriété est assurée par les lieurs d'interface de projection. Donc, en projetant un conteneur mémoire dans l'espace virtuel de différents processus via un lieur de projection, on obtient un mécanisme de mémoire virtuelle partagée (voir figure 9.9).

Lorsqu'un défaut de page se produit au sein d'un processus, le lieur d'interface détourne celui-ci vers les conteneurs. Le mécanisme de conteneur place une copie en mémoire locale tout en assurant le maintien de la cohérence. Enfin, le lieur d'interface projette la copie locale dans l'espace d'adressage du processus et fait coïncider les droits d'accès sur la page virtuelle aux droits de la page en conteneur.

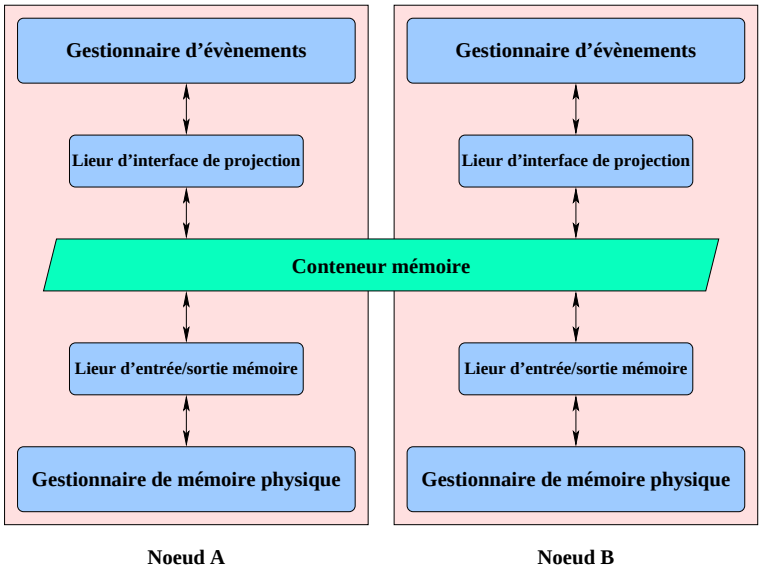

FIG. 9.9 – *Réalisation d'une mémoire virtuelle partagée fondée sur l'utilisation des conteneurs*

9.3.2 Projection de fichiers en mémoire

De manière similaire à la réalisation d'une MVP, il est possible de projeter un fichier dans une mémoire virtuelle en projetant un conteneur fichier dans l'espace virtuel d'un processus via un lieur de projection (voir figure 9.10). Il est également possible de projeter un fichier dans l'espace d'adressage d'un ou plusieurs processus, et de leurs threads associés. On obtient ainsi une projection de fichier en mémoire virtuelle partagée.

Nous distinguons deux types de projections : la projection partagée et la projection privée. Lorsqu'un fichier est projeté de manière partagée, toutes les modifications apportées à ce fichier sont immédiatement visibles par tous les processus projetant ce même fichier. Lorsqu'un fichier est projeté de manière privée, les modifications apportées à un fichier par un processus sont visibles par tous les « *threads* » de ce processus mais pas par les autres processus accédant à ce fichier.

9.3.2.1 Projection partagée

Une projection partagée correspond à la projection directe d'un conteneur fichier dans l'espace virtuel d'un processus via un lieur de projection. Toutes les modifications apportées à ce conteneur sont ainsi visibles par tous les « *threads* » et processus partageant ce même conteneur. Un conteneur fichier contenant toujours la version à jour d'un fichier, les modifications effectuées sur ce conteneur sont permanentes. Si ce conteneur est supprimé de la mémoire de la grappe, les modifications apportées à celui-ci sont répercutées sur l'unité de stockage hébergeant le fichier associé.

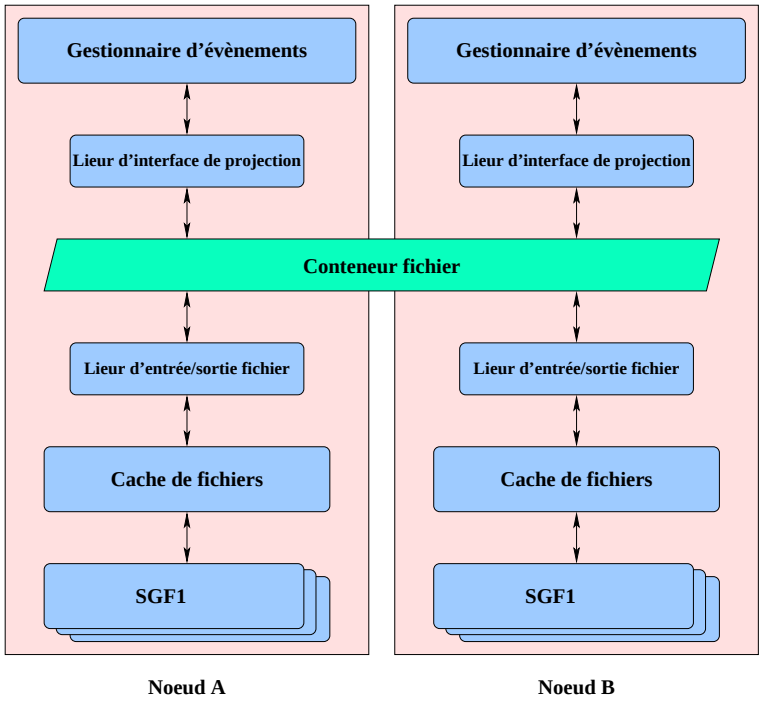

FIG. 9.10 – *Réalisation d'un mécanisme de projection partagée de fichier en mémoire fondé sur l'utilisation des conteneurs*

9.3.2.2 Projection privée

Une projection privée permet d'accéder en lecture et écriture à un fichier grâce à des accès mémoire sans reporter ces modifications sur le fichier. Pour cela, il n'est pas possible de projeter directement un conteneur fichier dans l'espace d'adressage d'un processus, sous peine de modifier son contenu.

La projection privée est réalisée en projetant un conteneur fichier dans un conteneur mémoire grâce à un lieur de conteneur, puis en projetant le conteneur mémoire dans l'espace d'adressage du processus grâce à un lieur de projection (voir figure 9.11). Il est ainsi possible d'accéder au fichier via la projection du conteneur fichier dans le conteneur mémoire et de modifier les données dans le conteneur mémoire sans modifier le conteneur fichier. Le conteneur mémoire étant projeté dans l'espace d'adressage d'un seul processus, les modifications sont visibles par tous les « *threads* » qu'il héberge (quelque soit leur localisation dans la grappe), mais pas par les autres processus.

9.3.3 Cache de fichiers coopératif

Aucun mécanisme particulier n'est à réaliser pour concevoir un cache de fichiers coopératifs. En effet, lorsqu'un fichier est placé en conteneur, celui-ci se comporte naturellement comme un cache coopératif de type cache coopératif distribué [30] :

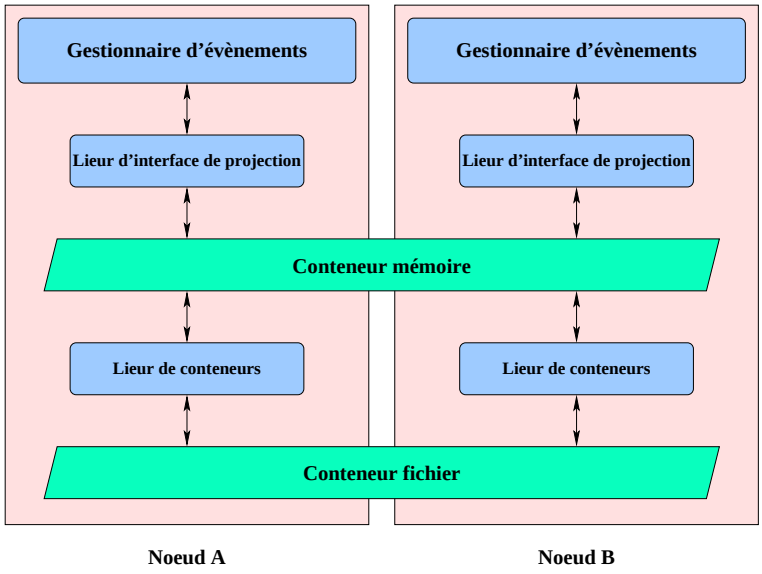

FIG. 9.11 – *Réalisation d'un mécanisme de projection privée de fichier en mémoire fondé sur l'utilisation des conteneurs*

- Un conteneur n'étant pas liée à un processus, il peut survivre à la mort d'un processus et conserver en mémoire les données chargées. Ces données peuvent alors être utilisées par d'autres processus sans nécessiter de chargement depuis un disque.
- Les données chargées par un nœud peuvent être utilisées par un autre nœud à travers un conteneur. Les données accédées par un processus sont recopiées automatiquement dans le cache de son nœud d'exécution par le mécanisme de conteneur.
- La cohérence des copies dans les différents caches est assurée automatiquement par le mécanisme de gestion de cohérence des conteneurs.
- La quantité de mémoire utilisée par le cache de fichiers s'adapte automatiquement en fonction de l'utilisation de la mémoire par les processus. Si un processus a des besoins importants en terme de consommation mémoire, les données en cache sont automatiquement supprimées au profit des données du processus. Les données du cache ne sont cependant pas perdues s'il en existe d'autres copies sur d'autres nœuds ou si elles peuvent être injectées.

9.3.4 Conception d'un système de fichiers réparti à base de conteneurs

Nous présentons dans ce paragraphe les concepts de base permettant de concevoir un système de gestion de fichiers distribué fondé sur l'utilisation des conteneurs. L'objectif de ce paragraphe n'est pas de détailler la conception et la réalisation d'un tel système, mais de présenter son architecture générale.

9.3.4.1 Objectif

L'objectif du système que nous souhaitons concevoir est d'offrir à l'utilisateur la vision d'un disque unique au dessus de l'ensemble des disques d'une grappe. Nous souhaitons obtenir les propriétés de transparence et d'indépendance à la localisation ainsi que les propriétés de partage logique et physique. Un fichier doit pouvoir être créé depuis n'importe quel nœud de la grappe et sauvegardé sur n'importe quel disque, quelque soit le processus à l'origine de la création du fichier et le répertoire contenant le fichier. Toute notion de localisation du fichier doit disparaître du nom et de l'interface d'accès au fichier. Enfin, nous ne souhaitons pas concevoir intégralement un SGF mais modifier légèrement un SGF existant, appelée **SGF hôte**, afin de répondre à l'un des objectifs de ce travail qui est de concevoir des services distribués en réutilisant au maximum les services existants.

9.3.4.2 Principe de fonctionnement

L'idée à la base de la conception de ce SGFD est de placer fichiers et répertoires en conteneur et de permettre ainsi leur consultation et modification depuis n'importe quel nœud de la grappe. A chaque entrée d'un fichier dans un répertoire est associé l'identifiant du nœud l'hébergeant sur disque, permettant de résoudre les problèmes liés à la conception d'un SGFD de la manière suivante :

Désignation : la désignation d'un fichier stocké sur les disques de la grappe est gérée de la même manière que dans un SGF traditionnel. Le mécanisme de conteneur offre l'illusion que les fichiers et les répertoires sont stockés sur un disque unique. Ainsi, une seule arborescence est utilisée, sans aucune contrainte de désignation liée à la localisation des fichiers sur les disques.

Localisation : la localisation d'un fichier en fonction de son nom externe (c'est-à-dire l'identification du nœud l'hébergeant) est réalisée grâce à un parcours de répertoire depuis la racine. Chaque sous-répertoire ainsi que le fichier lui-même sont localisés dans la grappe grâce à l'information de localisation associée à chaque entrée d'un répertoire.

Transparence : la seule information de localisation d'un fichier étant stockée dans la structure interne des répertoires, l'utilisateur n'a aucune connaissance de l'emplacement réel des fichiers et ne fait aucune distinction entre un fichier local et distant. Cette notion n'a d'ailleurs plus de sens pour lui.

Cohérence : la cohérence des accès concurrents est assurée par le cache de fichiers grâce au mécanisme de gestion de cohérence inclus dans le mécanisme de conteneur.

9.3.4.3 Gestion des répertoires

Au sein d'un système de fichiers de type Unix™, un répertoire est un fichier comme les autres. A ce titre, il est possible de placer ce fichier dans un conteneur et ainsi de permettre à n'importe quel nœud de la grappe de consulter le contenu du répertoire depuis n'importe

quel autre nœud sans avoir à concevoir et à programmer de mécanisme particulier. Lorsqu'un fichier est ajouté ou supprimé dans un répertoire sur un nœud, la modification du répertoire est immédiatement visible par tous les autres nœuds. Puisque les répertoires et les fichiers sont connus globalement grâce aux conteneurs, il est possible d'ajouter et de supprimer n'importe quel fichier de n'importe quel répertoire depuis n'importe quelle nœud, et ceci sans mise en œuvre spécifique. De plus, les fichiers peuvent être stockés physiquement sur n'importe quel disque de la grappe. Il est cependant nécessaire de mémoriser sur quel disque un fichier est stocké. Pour cela, l'identifiant du nœud hébergeant le fichier sur disque est ajouté à chaque entrée d'un répertoire. Cette information peut être ajoutée directement dans la structure de répertoire du SGF hôte, ou dans un fichier système associé à chaque répertoire. La première solution a l'inconvénient de rendre potentiellement le SGF modifié incompatible avec le SGF initial, du fait de la modification de la structure des répertoires.

Le répertoire racine est un cas particulier puisqu'il sert de point d'entrée dans le système de fichiers. A ce tire, il doit être commun à chaque nœud et dupliqué sur chaque disque de la grappe. Une fois le répertoire racine chargé en conteneur, la liste des sous-répertoires, fichiers ainsi que leur localisation physique est connu à travers la grappe.

9.3.4.4 Ouverture d'un fichier

Analysons le cas de l'ouverture d'un fichier qui n'est pas encore placé en conteneur et dont tous les répertoires composant son chemin d'accès sont chargés en mémoire au sein de conteneurs. L'ouverture de ces répertoires est réalisée récursivement de la même manière.

Le système de fichiers commence par déterminer quel conteneur contient le répertoire hébergeant le fichier. Pour cela, un parcours du chemin d'accès au fichier est réalisé depuis la racine du système de fichier. Une fois le conteneur identifié, le système de fichier recherche l'entrée correspondant au fichier dans le répertoire associé au conteneur. De cette entrée le SGF détermine sur quel nœud est stocké le fichier et crée un conteneur fichier attaché à ce nœud. Le fichier est alors visible est accessible sur tous les nœuds de la grappe à travers ce conteneur.

9.3.4.5 Propriétés du système

Le système que nous venons de présenter répond aux critères d'un système à image unique, puisqu'il remplit les propriétés suivantes :

- **Transparence à la localisation** : la localisation physique des fichiers est stockée dans la structure de répertoire, sans aucune répercussion sur le nom des fichiers. On obtient ainsi une transparence totale à la localisation physique des fichiers.
- **Indépendance de localisation** : le déplacement physique d'un fichier du disque d'un nœud vers un autre nœud consiste à déplacer physiquement les données et mettre à jour l'information de localisation dans la structure de répertoire. Si un conteneur associé au fichier déplacé est chargé en mémoire, son attachement doit être modifié. Ce déplacement est totalement transparent à l'utilisateur : le nom du

fichier et les méthodes d'accès à celui-ci ne sont pas modifiées.
- **Partage logique** : lorsqu'un fichier stocké sur un disque particulier de la grappe est ouvert, il est placé en conteneur et donc accessible par tous les nœuds de la grappe.
- **Partage physique** : lorsqu'un processus crée un nouveau fichier, celui-ci est stocké par défaut sur le disque du nœud d'exécution du processus. Si ce disque est plein, le fichier peut être stocké sur un autre disque, en modifiant simplement l'attachement du conteneur associé.
- **Extensibilité des performances** : lorsqu'un nœud est ajouté à la grappe, son disque local peut être utilisé pour stocker des fichiers, augmentant alors la capacité totale de stockage et la bande passante totale théorique. Un bon équilibrage des fichiers sur les disques permet en effet de distribuer la charge des accès disques et d'augmenter ainsi la performante globale des accès aux fichiers.
- **Transparence à l'extensibilité** : lorsqu'un nœud est ajouté à la grappe, la liste des nœuds appartenant à la grappe est mise à jour, permettant ainsi au système de fichiers d'utiliser les nouvelles ressources offertes par ce nœud.

9.3.4.6 Problèmes de tolérance aux fautes

Dans le système décrit précédemment, les propriétés de haute disponibilité et de tolérance aux fautes ne sont pas assurées. En effet, la défaillance d'un nœud cause la perte des fichiers stockés sur ce nœud et plus grave, la perte des répertoires y étant hébergés. La perte d'un répertoire provoque la perte de toute la sous-arborescence associée.

Duplication des répertoires Les répertoires étant des structures très sensibles dans un système de fichiers, il est essentiel de pouvoir continuer à y accéder même après une défaillance importante, par exemple la défaillance simultanée et permanente de plusieurs nœuds.

Afin d'éviter la perte des données d'un répertoire, nous dupliquons les répertoires sur tous les disques de la grappe. Ainsi, quelque soit le nombre de nœuds défaillants (à la condition qu'il subsiste au moins un nœud valide), il est toujours possible d'accéder à la totalité de l'arborescence du système de fichiers.

Les données associées à un répertoire étant de taille modeste (de quelques octets à quelques kilo-octets), cette approche est raisonnable en terme de consommation d'espace disque. De plus, la grande sensibilité de ces données justifie amplement ce choix de conception.

Redondance des fichiers La perte d'un fichier de données n'est pas critique pour le fonctionnement du système de fichiers. Cependant, cette perte peut être préjudiciable pour l'utilisateur. Afin d'éviter la perte de fichiers de données, nous utilisons un mécanisme de type RAID logiciel de premier niveau. Pour cela, chaque fichier est stocké sur deux disques situés sur deux nœuds différents de la grappe. Ainsi, en cas de défaillance d'un nœud, les fichiers stockés sur ce nœud sont toujours accessibles sans aucune dégradation de performance, grâce aux copies présentes sur le disque miroir.

Les fichiers très sensibles peuvent être dupliqués sur tous les nœuds de la grappe, à la manière des répertoires, afin de tolérer la défaillance d'un grand nombre de nœuds. Cette solution très coûteuse en terme d'espace de stockage est à réserver aux fichiers systèmes critiques et aux fichiers de données sensibles.

9.4 La migration de processus dans le cadre d'un système à base de conteneurs

La migration d'un processus d'un site d'exécution vers un autre consiste principalement en deux phases : (1) déplacer le contexte d'exécution du processus et (2) assurer au processus migré un accès transparent aux ressources qu'il utilisait sur le nœud initial. Nous présentons dans cette partie comment les conteneurs permettent de maintenir le lien entre un processus et la plupart des ressources situées sur le nœud source, sans nécessiter la conception de mécanismes spécialisés.

9.4.1 Migration de l'espace d'adressage

Prenons l'exemple d'un processus dont l'espace d'adressage est composé des segments de mémoire virtuelle suivants :

- **Segment de texte**. Ce segment correspond à la projection du fichier contenant le code de l'application dans l'espace d'adressage du processus. Ce segment est en lecture seule.
- **Segment de données initialisées**. Ce segment contient les variables dont la valeur est initialisée au sein du fichier source, avant même la compilation. Il correspond à la projection dans l'espace d'adressage du processus, du fichier texte de l'application avec un décalage plaçant le début de la projection sur la zone du fichier contenant les données initialisées. Ce segment est accessible en lecture et en écriture. Cependant, les écritures ne sont pas répercutées sur le fichier associé.
- **Segment de données non initialisées**. Ce segment contient les variables du programme initialisées durant l'exécution et correspond à une zone de mémoire physique projeté en mémoire virtuelle. Ce segment est accessible en lecture et en écriture.
- **Segment de projection de fichier**. Ce segment correspond à un fichier local projeté dans l'espace d'adressage du processus. Ce segment est accessible en lecture et en écriture. Les modifications apportées à ce segment sont sauvegardées dans le fichier associé.
- **Segment de pile**. Ce segment correspond à un segment de mémoire. Il est accessible en lecture et en écriture.

Lorsque le processus est migré, les segments du processus n'étant pas encore liés à un conteneur sont associés à de nouveaux conteneurs. C'est notamment le cas pour les segments de mémoire. Tout fichier étant dans un conteneur, les segments associés à un fichier sont déjà placés en conteneur. Pour chaque segment de mémoire, un nouveau conteneur est créé et associé au segment (voir figure 9.12).

Lors de la migration du processus, le mécanisme de migration de contexte déplace vers le nœud cible l'image mémoire du processus, c'est-à-dire la liste de ses segments, leur taille, adresse de début et de fin et les conteneurs associés. En revanche, aucune donnée contenue dans les segments n'est migrée.

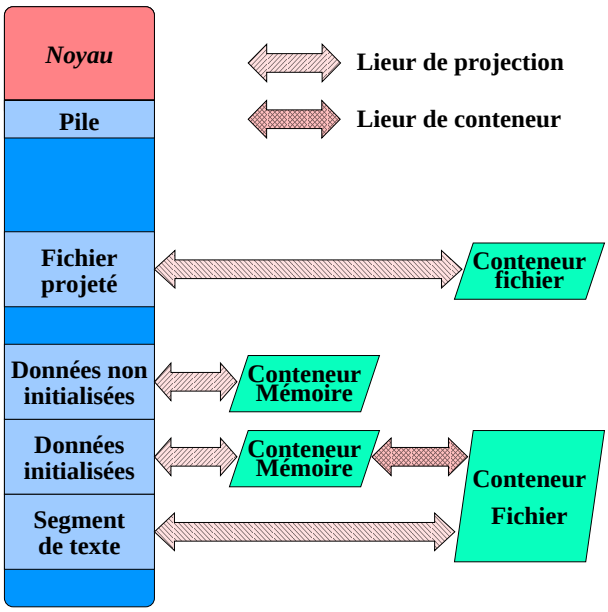

FIG. 9.12 – *Liaison des segments de mémoire virtuelle d'un processus à un ensemble de conteneurs*

Après migration du processus, l'accès dans l'espace d'adressage du processus à des pages invalides provoque un défaut de page, détourné vers un lieur de projection qui assure la migration des données correspondantes depuis la machine source vers la machine cible à travers les conteneurs associés. Dans le cas des segments de fichiers projetés, les données peuvent déjà se trouver sur la machine cible grâce au comportement de cache coopératif des conteneurs fichiers. Dans ce cas, aucune transaction réseau n'est nécessaire.

9.4.2 Migration de fichiers ouverts

Supposons que notre processus exemple ait ouvert un fichier local grâce à l'interface standard d'accès aux fichiers (ouverture, lecture, écriture). Lors de l'ouverture du fichier, le conteneur hébergeant le fichier est associé à la représentation en mémoire de l'i-noeud du fichier.

Lors de la migration de processus, le mécanisme de migration de contexte déplace vers le nœud cible la liste des descripteurs des fichiers ouverts par le processus ainsi que les identifiants des conteneurs associés. Sur le nœud cible, la liaison entre chaque descripteur de fichier du processus migré et les conteneurs associés est rétablie.

Après migration du processus, l'accès aux fichiers ouverts par le processus sur la machine source est réalisé de manière transparente par l'intermédiaire des fonctions de lecture et écriture. Ces accès sont détournés grâce aux lieurs de fichier vers les conteneurs associés. Les données situées dans le cache de fichier de la machine source sont recopiées automatiquement dans le cache local de la machine cible grâce aux conteneurs. Des données des fichiers ouverts par le processus peuvent déjà être présents sur la machine cible grâce au comportement naturellement coopératif des conteneurs fichiers.

9.4.3 Discussion

Dans le contexte de la migration de processus, les conteneurs permettent de réaliser une partie difficile de la migration qui est de maintenir le lien entre le processus migré et les données laissées sur la machine source. Des problèmes aussi difficiles que la migration de l'espace d'adressage et le maintien des liens avec les fichiers ouverts sont résolus très simplement sans aucun mécanisme particulier. La migration est alors réellement légère puisqu'elle consiste simplement en la migration du contexte du processus.

Les conteneurs ne règlent cependant pas tous les problèmes liés à la migration de processus. Tout d'abord, la migration du contexte n'est pas une opération légère, puisqu'elle nécessite le déplacement de nombreuses structures de données situées dans le noyau et le rétablissement de liaisons entre les structures représentant le processus migré et les structures associés aux périphériques, autres processus, etc. Pour cette partie, les conteneurs ne sont d'aucun secours, puisqu'il serait totalement déraisonnable de placer en conteneur les structures de données représentant un processus.

D'autre part, les conteneurs ne permettent pas de partager des flux de données à travers une grappe. Il est ainsi impossible d'associer un clavier, une souris ou une imprimante à un conteneur. Seuls les périphériques de type *bloc* peuvent être gérés grâce aux conteneurs. Ainsi, lors de la migration d'un processus, le lien entre le processus migré et les périphériques de type *caractère* situés sur le nœud source doit toujours être maintenu grâce à des mécanismes spécialisés.

Enfin, la gestion de la filiation entre le processus migré, son père et ses fils, la gestion des signaux et les autres mécanismes intervenant directement sur le processus et ses structures de données posent des problèmes difficiles à résoudre de manière satisfaisante. A nouveau, les conteneurs sont inefficaces pour résoudre ces problèmes.

En conclusion, les conteneurs permettent de simplifier de manière élégante et efficace une partie des problèmes posés par la migration de processus, mais de nombreuses difficultés subsistent pour lesquelles les conteneurs n'apportent pas de solution satisfaisante. Cependant, une piste de recherche intéressante serait l'extension du concept de conteneur pour la gestion des périphériques de type *caractère*. Ceci permettrait de simplifier encore la migration de processus et l'accès aux périphériques distants.

9.5 Résumé

Nous avons consacré ce chapitre à la conception des lieurs et à la réalisation de services systèmes fondés sur l'utilisation de conteneurs.

Nous avons présentés la conception de deux types de lieurs : les lieurs mémoires et les lieurs fichiers. Le lieur d'interface mémoire propose une interface permettant de projeter un conteneur dans l'espace d'adressage d'un processus. Le lieur d'interface fichier permet d'accéder à un conteneur via l'interface standard d'accès aux fichiers, de type lecture/écriture. D'autres types de lieur d'interface peuvent être créés afin d'interfacer les conteneurs avec d'autres parties du système d'exploitation hôte.

Le lieur d'entrée/sortie mémoire permet d'allouer et de libérer des cadres de page permettant d'instancier un conteneur en **conteneur mémoire**. Le lieur d'entrée/sortie fichier permet de lire et d'écrire des pages de données dans un fichier, permettant d'instancier un conteneur en **conteneur fichier**. D'autres types de lieurs d'entrée/sortie peuvent être réalisés permettant d'instancier de nouveaux types de conteneurs.

Disposant des mécanismes de conteneurs et de lieurs, nous avons ensuite présentés comment les conteneurs peuvent être utilisés pour réaliser des services systèmes de haut niveau. Une mémoire virtuelle partagée est obtenue en projetant un conteneur mémoire dans l'espace d'adressage de plusieurs processus. Un cache de fichier coopératif est obtenu simplement en plaçant chaque fichiers ouverts dans un conteneur. Un mécanisme de projection de fichier en mémoire et en MPRL est obtenu en projetant un conteneur fichier dans l'espace d'adressage d'un ou plusieurs processus. Enfin, en plaçant les répertoires d'un SGF en conteneur, il est possible d'obtenir simplement un SGFD remplissant les propriétés d'un système à image unique.

Enfin, nous avons montré comment les conteneurs peuvent simplifier les mécanismes de migration de processus en assurant automatiquement la liaison entre le processus migré et les périphériques qu'il utilisait sur son nœud d'exécution initial. Les conteneurs ne résolvent cependant pas tous les problèmes liés à la migration. Un extension des conteneurs permettant de partager des périphériques de type caractère permettrait de simplifier encore la migration de processus.

Troisième partie

Mise en oeuvre et évaluation

10 ÉLÉMENTS DE MISE EN OEUVRE DES CONTENEURS AU SEIN DU SYSTÈME D'EXPLOITATION GOBELINS

Nous avons présenté dans la partie II, la conception d'un mécanisme logiciel s'intégrant au sein d'un système d'exploitation hôte s'exécutant sur une grappe afin d'offrir un système à image unique. Nous présentons dans ce chapitre la mise en œuvre de ce mécanisme au sein du système d'exploitation LINUX. Nous avons baptisé le système à image unique obtenu GOBELINS.

Nous démontrons dans cette partie que le mécanisme de conteneur permet de réaliser des services systèmes distribués avec une mise en œuvre très légère. La mise en œuvre des conteneurs représente 1700 lignes de code et la mise en œuvre des lieurs représente 1100 lignes de code. Enfin, seules 138 lignes ont été ajoutées ou modifiées dans le noyau du système LINUX 2.2.13.

10.1 Plate-forme d'expérimentation

10.1.1 La grappe PARASKI

L'architecture cible sur laquelle GOBELINS est mis en œuvre est la grappe de calcul PARASKI. Cette grappe est composée de 40 PCs bi-processeurs de générations différentes. Les nœuds les plus anciens sont équipés de processeurs Pentium Pro cadencés à 200 MHz et de 128 Mo de mémoire centrale, tandis que les nœuds les plus -récents sont équipés de processeurs Pentium III cadencés à 1 GHz et de 512 Mo de mémoire centrale.

Quatre technologies d'interconnexion sont utilisées : Fast Ethernet, Gigabit Ethernet, Myrinet et SCI. Le réseau Fast Ethernet est présent sur tous les nœuds, tandis que les autres technologies de communication équipent des sous-grappes à des fins d'expérimentation.

10.1.2 Le système d'exploitation LINUX

Le système d'exploitation hôte choisi pour la mise en œuvre du système GOBELINS est GNU/LINUX. Le système d'exploitation Linux est un système d'exploitation de type Unix™développé par la communauté du logiciel libre. Fondé sur un noyau monolithique, il inclut la notion de **module** permettant d'inclure de nouvelles fonctionnalités au sein du

noyau durant son exécution. GOBELINS a été mis en œuvre par l'intermédiaire de modules intégrés au noyau LINUX.

Dans ce chapitre, nous supposons que le lecteur a une bonne connaissance du fonctionnement interne du système LINUX. Le lecteur non averti pourra trouver en annexe D un rapide aperçu de LINUX fournissant les bases nécessaires à la compréhension de ce chapitre.

10.2 Gobelins : un système d'exploitation pour grappe de calcul fondé sur le concept de conteneurs

GOBELINS est un système d'exploitation pour grappe de calculateurs fondé sur le concept de conteneurs. L'objectif de ce système est d'offrir un système à image unique satisfaisant toutes les propriétés énoncées dans le chapitre 2.3 et ce, pour toutes les ressources de la grappe.

10.2.1 Architecture générale du système GOBELINS

La mise en œuvre de GOBELINS repose principalement sur trois modules : le module de communications baptisé GIMLI, le module de gestion des conteneurs baptisé GANDALF et le module de migration de processus baptisé ARAGORN. Ces trois modules sont chargés au sein du noyau LINUX afin d'y intégrer les fonctionnalités fournies par GOBELINS (voir figure 10.1).

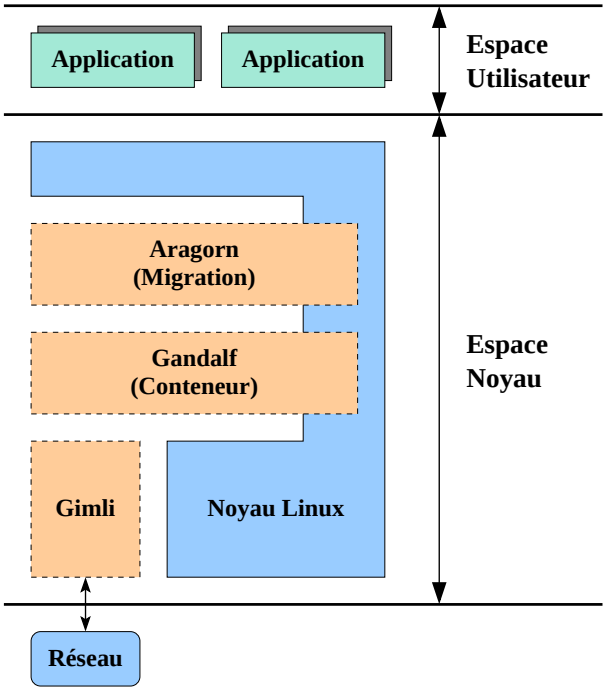

FIG. 10.1 – *Architecture générale du système Gobelins*

Le module GIMLI met en œuvre une bibliothèque de communication haute performance permettant d'envoyer et de recevoir des messages de noyau à noyau. Le module GANDALF met en œuvre les conteneurs et les lieurs mémoires et fichiers. Enfin, le module ARAGORN réalise la migration de processus et utilise les conteneurs pour la migration de l'espace d'adressage et des fichiers ouverts.

10.2.2 Communications haute performance : le module Gimli

La mise en œuvre des conteneurs au sein du système LINUX ainsi que la réalisation des autres mécanismes du système GOBELINS (migration de processus, haute disponibilité, etc) nécessite de pouvoir échanger des données entre les noyaux s'exécutant sur les différents nœuds de la grappe. Ceci implique l'utilisation de mécanismes de communication dans le noyau. Linux offre deux types de mécanismes de communication au sein du noyau : les « *sockets* » et les RPC. Or, ces deux mécanismes offrent des performances médiocres et une interface de programmation complexe. C'est pourquoi le premier objectif de la mise en œuvre a été de doter le système GOBELINS d'une bibliothèque de communication à haute performance et offrant une interface de programmation simple. Cette bibliothèque a été mise en œuvre au sein du module GIMLI (GOBELINS Interactive Message LIbrary) et répond aux objectifs suivants :

- **Haute performance** : la bibliothèque doit offrir des latences faibles et un débit élevé ;
- **Indépendance vis-à-vis du matériel** : la bibliothèque doit fonctionner de la même manière et avec la même interface quelque soit la technologie réseau sous-jacente ;
- **fiabilité** : tout message envoyé doit être transmis sans aucune perte ou altération de données ;
- **Interface simple** : l'interface d'utilisation doit être simple et adaptée à la programmation distribuée.

Le module GIMLI ne faisant pas parti du cœur de ce travail, sa mise en œuvre ainsi que son étude de performance sont présentées en annexe E.

10.2.3 Gestion globale de la mémoire : le module Gandalf

Le module GANDALF met en œuvre les conteneurs et les lieurs présentés dans ce document. L'objectif de cette mise en œuvre est de valider le concept de conteneur et de démontrer comment il est possible de mettre en œuvre des mécanismes systèmes distribués complexes avec un minimum de modifications d'un système d'exploitation hôte.

Au sein du noyau LINUX, il n'est pas prévu de pouvoir accéder à toutes les variables et de pouvoir modifier toutes les fonctions depuis un module. C'est pourquoi la mise en œuvre des conteneurs et des lieurs a nécessité de modifier quelques lignes de code dans le noyau (138 lignes) afin d'autoriser l'accès et la modification des variables et fonctions nécessaires à la réalisation du module GANDALF. Les détails de mise en œuvre de ce module sont présentés dans la suite de ce chapitre.

10.2.4 Gestion globale des processus : le module Aragorn

Le module ARAGORN met en œuvre les mécanismes de création de processus à distance et de migration de processus. Comme nous l'avons déjà évoqué dans le paragraphe 5.1.2, la migration de processus consiste en deux phases principales : (1) la migration du contexte d'exécution du processus et (2) le maintien des liens entre le processus migré et les ressources qu'il utilisait sur son nœud d'exécution initial. Le module ARAGORN réalise la première partie de ce travail, la seconde étant assurée automatiquement par les mécanismes de conteneurs et de lieurs du module GANDALF. Dans la suite de ce paragraphe, nous décrivons brièvement le fonctionnement de ce module. De plus amples informations peuvent être trouvées dans [94].

10.2.4.1 Migration du contexte d'exécution

Avant la migration d'un processus, le module ARAGORN parcourt la liste des segments mémoire du processus (appelés VMAs dans le noyau LINUX) et recherche les VMAs qui ne sont pas encore liées à un conteneur. Pour chacune de ces VMA, un nouveau conteneur est créé et lui est associé.

ARAGORN extrait ensuite le contexte du processus contenu dans la structure *task_struct* et les structures liées à ce processus qui ne peuvent être reconstruites sur le nœud cible. Ce contexte est envoyé au nœud cible qui le place en mémoire locale. Cette structure incomplète est passée en argument à la fonction *do_fork* du noyau qui duplique ce processus fantôme pour en faire un véritable processus exécutable par le noyau local. La fonction *do_fork* a été légèrement modifiée pour pouvoir réaliser la liaison entre le processus créé et les conteneurs. Cette liaison consiste à associer à chaque VMA du processus créé et à chaque fichier ouvert, le conteneur auquel il était lié sur le nœud initial. Enfin, les données associées au processus fantôme sont détruites et le processus situé sur le nœud initial est arrêté.

10.2.4.2 Conservation des liens avec le nœud initial

Lorsque le processus migré redémarre sur le nœud cible, aucune page de son espace d'adressage n'est présente sur son nouveau nœud d'exécution. Les pages mémoires nécessaires à son fonctionnement sont migrées automatiquement grâce au mécanisme de conteneur. Les pages provenant d'un fichier peuvent déjà être présentes dans le cache local du nœud grâce aux conteneurs fichiers. Dans ce cas, aucune migration de données n'est nécessaire.

L'accès aux fichiers ouverts sur le nœud source est également réalisé à travers les conteneurs. Lors de chaque accès à un fichier, si la donnée n'est pas présente dans le cache local, les conteneurs se chargent d'en obtenir une copie depuis le site d'exécution initial ou depuis le serveur NFS.

10.3 Mise en œuvre des conteneurs

10.3.1 Architecture générale

La mise en œuvre des conteneurs repose sur l'utilisation de quatre entités logicielles (voir figure 10.2) : l'interface d'accès aux conteneurs, le gestionnaire de conteneurs, le gestionnaire de pages et le serveur de pages. Le gestionnaire de conteneur permet la création et la destruction de conteneurs. Le gestionnaire de pages est une mise en œuvre distribuée d'un répertoire mémorisant la localisation des copies maîtres. Enfin, le serveur de pages permet l'envoi et l'invalidation de pages.

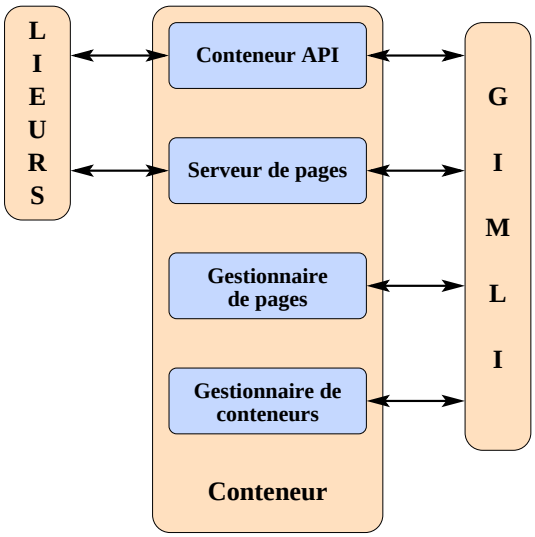

FIG. 10.2 – *Architecture logicielle des conteneurs*

10.3.2 Principales structures de données

10.3.2.1 Structures de données utilisées pour la mise en œuvre des conteneurs

La figure 10.3 présente un aperçu des structures de données utilisées pour la mise en œuvre des conteneurs. Un tableau nommé *container* **[1]** dupliqué sur tous les nœuds de la grappe contient la liste des conteneurs existants dans le système. Chaque conteneur est représenté par une structure de donnée nommée *container_t* **[2]** dont les champs sont présentés sur la figure 10.4. Cette structure contient en particulier un pointeur vers la liste des lieurs d'interface connectés au conteneur **[3]**, un pointeur vers le lieur d'entrée/sortie associé au conteneur **[4]** ainsi que deux tables : *page_table* et *page_info*.

Le champs *page_table* est un pointeur vers la table des pages du conteneur **[5]**. A chaque page d'un conteneur est associée une structure nommée *ctnr_Page_t* (voir figure 10.5) indiquant l'état de la page au sens du protocole de cohérence, l'adresse de la page en mémoire physique, ainsi que la liste des copies lorsque la page est la copie maître (le

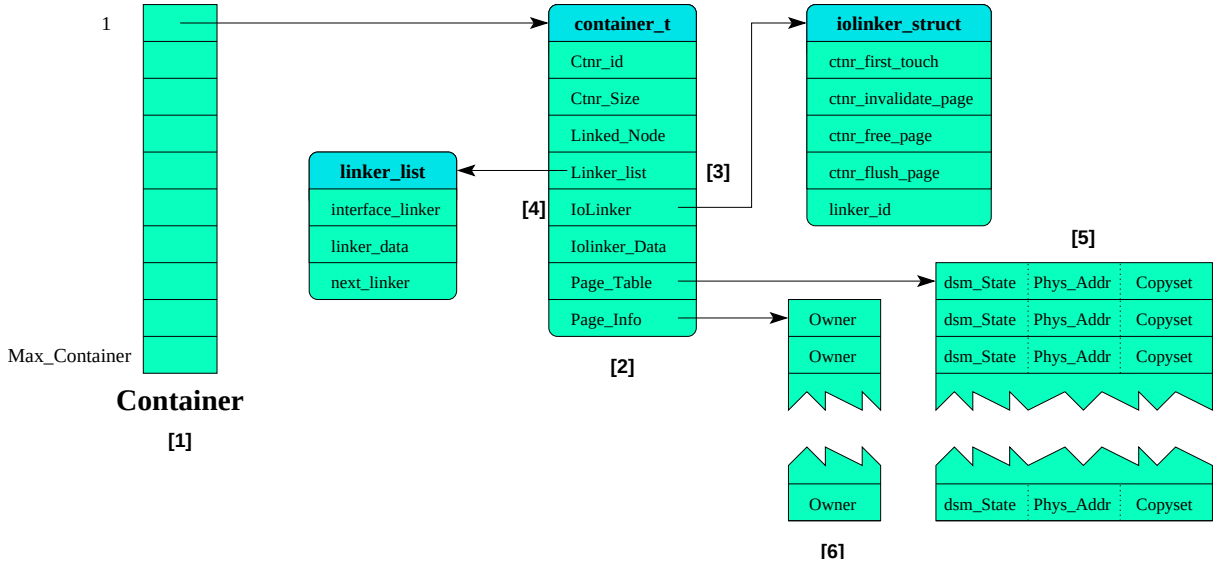

FIG. 10.3 – *Structures de données utilisées pour la gestion des conteneurs*

copyset). Le *copyset* n'est géré que sur le nœud propriétaire. Sur les autres nœuds, ce champs est inutilisé.

Le champs *page_info* est un pointeur vers une table distribuée sur les nœuds de la grappe et contenant pour chaque page l'identité du nœud propriétaire **[6]**. Cette table est gérée par le gestionnaire de pages.

```
Structure container_t :
{
 ctnr_Id          ; { Identifiant du conteneur }
 linked_node      ; { Identifiant du nœud lié au conteneur }
 ctnr_size        ; { Taille du conteneur en nombre de pages }
 page_table       ; { Table des pages du conteneur (table locale) }
 page_info        ; { Table du gestionnaire de pages (table globale) }
 linker_list      ; { Liste des lieurs d'interface connectés au conteneur }
 iolinker         ; { Lieur d'entrée/sortie associé au conteneur }
 iolinker_data    ; { Données privées du lieur d'entrée/sortie associé au conteneur }
 offset           ; { Déplacement du conteneur dans le périphérique lié }
 flags            ; { Paramètres du conteneur }
 }
```

FIG. 10.4 – Structure représentant un conteneur

```
Structure ctnr_Page_t :
{
 dsm_State   ; { État de la page au sens du protocole de cohérence }
 phys_Addr   ; { Adresse de la page en mémoire physique }
 copyset     ; { Liste des copies de la page dans la grappe }
 }
```

FIG. 10.5 – Structure représentant une page d'un conteneur

10.3.2.2 Structures de données utilisées pour la mise en œuvre des lieurs

Chaque lieur d'entrée/sortie met en œuvre les fonctions *first_touch*, *invalidate_page*, *free_page* et *flush_page* pour le type de périphérique dont il a la charge (mémoire, disque, etc). Ces fonctions sont répertoriées au sein de la structure *iolinker_struct* qui représente un lieur d'entrée/sortie (voir figure 10.6). Cette structure contient les pointeurs sur les fonctions d'interface du lieur et un identifiant de lieur.

Chaque lieur d'interface est représenté par la structure *interface_linker_t* (voir figure 10.7). Cette structure contient un pointeur vers la fonction *change_access* mise en œuvre par le lieur et son identifiant.

```
Structure iolinker_struct :
{
 ctnr_first_touch       ; { Pointeur vers la fonction de création de page sur premier accès }
 ctnr_invalidate_page   ; { Pointeur vers la fonction d'invalidation de page }
 ctnr_free_page         ; { Pointeur vers la fonction de libération d'une page }
 ctnr_flush_page        ; { Pointeur vers la fonction de remplacement de page }
 linker_id              ; { Identifiant du lieur }
 }
```

FIG. 10.6 – Structure représentant un lieur d'entrée/sortie

```
Structure interface_linker_t :
{
 change_access   ; { Pointeur vers la fonction de changement de droit d'accès }
 linker_id       ; { Identifiant du lieur }
 }
```

FIG. 10.7 – Structure représentant un lieur d'interface

Un conteneur peut être lié à plusieurs services systèmes et à plusieurs instances d'un même service. Par exemple, un conteneur peut être projeté dans l'espace d'adressage de

plusieurs processus différents sur un même nœud. Chaque projection correspond à une liaison différente.

Chaque liaison est représentée par la structure *linker_list* (voir figure 10.8). Cette structure contient un pointeur vers le lieur d'interface utilisé pour la liaison, un pointeur vers des données privées de ce lieur et un pointeur vers la liaison suivante.

```
Structure linker_list :
{
 interface_linker   ; { Lieur d'interface utilisé pour cette liaison }
 linker_data        ; { Données du lieur associées à la liaison }
 next_linker        ; { Liaison suivante suivant }
 }
```

FIG. 10.8 – Structure représentant la liste des liaisons d'interface associées à un conteneur

10.3.3 Mise en œuvre de l'interface des conteneurs

L'interface des conteneurs consiste en six fonctions :
- **create_new_container** : Création d'un nouveau conteneur.
- **free_container** : Destruction d'un conteneur.
- **ctnr_find_page** : Vérification de la présence d'une page d'un conteneur en mémoire locale.
- **ctnr_get_page** : Place en mémoire centrale une copie en lecture d'une page d'un conteneur.
- **ctnr_grab_page** : Place en mémoire centrale une copie en écriture d'une page d'un conteneur.
- **ctnr_flush_page** : Suppression d'une page d'un conteneur de la mémoire centrale.

La suite de ce paragraphe présente la mise en œuvre de ces fonctions.

10.3.3.1 Création d'un conteneur : la fonction *create_new_container*

Lors d'un appel à la fonction *create_new_container*, une entrée libre dans la table des conteneurs est recherchée. Un identifiant unique de conteneur est créé est stocké dans la structure. Les différents champs de cette structure sont initialisés, notamment la taille du conteneur, le nœud attaché, le lieur d'entrée/sortie associé et le déplacement du début du conteneur par rapport au début de l'objet associé au conteneur (fichier, autre conteneur).

Une requête de création de conteneur (*REQ_CTNR_CREATE*) est diffusée à l'ensemble des nœuds de la grappe et les acquittements correspondants sont attendus. Enfin, l'adresse de la structure de conteneur créée est retournée.

10.3.3.2 Libération d'un conteneur : la fonction *free_container*

Lors de l'appel à la fonction *free_container*, une requête de libération de conteneur (*REQ_CTNR_FREE*) est envoyée à tous les nœuds de la grappe. Enfin, la structure

locale est libérée.

10.3.3.3 Vérification de la présence d'une page en mémoire locale : la fonction *ctnr_find_page*

La fonction *ctnr_find_page* cherche l'entrée correspondant au conteneur passé en argument et retourne l'adresse physique de la page demandée. Cette adresse est stockée dans la structure *ctnr_page_t* et vaut *NULL* si la page n'est pas présente en mémoire locale.

10.3.3.4 Obtention d'une copie en mémoire locale : la fonction *ctnr_get_page*

Lorsqu'un appel à la fonction *ctnr_get_page* est effectué, une requête de copie en lecture (*REQ_PM_PAGE_READ*) est envoyée au nœud gestionnaire de la page. Celui-ci est déterminé grâce à un modulo réalisé entre le numéro de la page demandée et le nombre de nœuds de la grappe. Un message contenant une copie de la page est ensuite attendue. Si le message reçu n'est pas la page attendu, mais une requête de création de page, la fonction *ctnr_first_touch* du lieur d'entrée/sortie est appelée afin de charger la page depuis le périphérique associé, puis le nœud local est ajouté au *copyset*.

L'état de la page dans la table des pages du conteneur est placé à lecture seule et l'adresse de la page est retournée.

10.3.3.5 Obtention d'une copie en écriture en mémoire locale : la fonction *ctnr_grab_page*

Lors d'un appel à la fonction *ctnr_grab_page*, si le nœud local est le propriétaire de la page, des requêtes d'invalidation de page (*REQ_PS_PAGE_INVALID*) sont envoyées à tous les nœuds présents dans le *copyset* et les acquittements correspondants sont attendus.

Si le nœud local n'est pas le propriétaire de la page, une requête de copie en écriture (*REQ_PM_PAGE_WRITE*) est envoyée au nœud gestionnaire de la page. Un message contenant une copie de la page est ensuite attendu. Si le message reçu n'est pas la page attendu, mais une requête de création de page, la fonction *ctnr_first_touch* du lieur d'entrée/sortie est appelée afin de charger la page depuis le périphérique associé, puis le nœud local est ajouté au *copyset*.

L'état de la page dans la table des pages du conteneur est placé à lecture/écriture et l'adresse de la page est retournée.

10.3.3.6 Suppression d'un cadre de page de la mémoire centrale : la fonction *ctnr_flush_page*

Cette fonction n'a pas été mise en œuvre.

10.3.4 Mise en œuvre du gestionnaire de conteneur

Le gestionnaire de conteneurs est un *thread* noyau qui assure la création, l'initialisation et la connexion aux conteneurs. Il peut recevoir 2 types de requêtes :

- **REQ_CTNR_CREATE** : demande de création de conteneur ;
- **REQ_CTNR_FREE** : demande de libération de conteneur.

Sur une demande de création de conteneur, le gestionnaire recherche une structure de conteneur libre, l'initialise et lui affecte l'identifiant de conteneur envoyé par le nœud à l'initiative de la création.

Sur une demande de libération de conteneur, le gestionnaire recherche le conteneur concerné dans sa table des conteneurs et appelle la fonction de libération de conteneur. Celle-ci libère les structures associées et place le conteneur dans la liste des conteneurs libres.

10.3.5 Mise en œuvre du gestionnaire de pages

Le gestionnaire de page est un *thread* noyau qui mémorise la localisation des propriétaires des pages et relaie les requêtes qui leur sont destinées depuis un nœud en défaut de page. Il peut recevoir deux types de requêtes :

- **REQ_PM_PAGE_READ** : demande de copie en lecture ;
- **REQ_PM_PAGE_WRITE** : demande de copie en écriture.

10.3.5.1 Copie en lecture

Sur une demande de copie en lecture, le gestionnaire vérifie si la page a un propriétaire grâce au champs *owner* de la table *pageInfo* de la structure de conteneur. Si la page a un propriétaire, le gestionnaire lui transmet une requête de copie en lecture (*REQ_PS_PAGE_READ*). Dans le cas contraire, si le conteneur n'est pas attaché ou que le nœud attaché est le nœud demandeur, le gestionnaire note que le nœud demandeur est le nouveau propriétaire et envoie à celui-ci une requête de création de page. Enfin, si la page n'a pas de propriétaire et que le conteneur est attaché à un nœud autre que le nœud demandeur, le gestionnaire note que le nouveau propriétaire est le nœud attaché et envoie une requête de création de page en lecture (*REQ_PS_FIRST_TOUCH_READ*) à celui-ci.

10.3.5.2 Copie en écriture

Sur une demande de copie en écriture, le gestionnaire vérifie si la page a un propriétaire. Si la page a un propriétaire, le gestionnaire lui envoie une requête de copie en écriture (*REQ_PS_PAGE_WRITE*), note que le nouveau propriétaire est le nœud demandeur et attend un acquittement du nœud demandeur. Si le conteneur n'est pas attaché ou que le nœud attaché est le nœud demandeur, le gestionnaire note que le nœud demandeur est le nouveau propriétaire et envoie à celui-ci une requête de création de page. Enfin, si la page n'a pas de propriétaire et que le conteneur est attaché à un nœud autre que le nœud

demandeur, le gestionnaire note que le nouveau propriétaire est le nœud demandeur et envoie une requête de création de page en écriture (*REQ_PS_FIRST_TOUCH_WRITE*) au nœud attaché.

10.3.6 Mise en œuvre du serveur de pages

Le serveur de page est un *thread* noyau qui assure l'envoi, l'invalidation et la création sur premier accès de pages. Il peut recevoir cinq types de requêtes :

- **REQ_PS_PAGE_INVALID** : requête d'invalidation de page.
- **REQ_PS_PAGE_READ** : requête de copie en lecture.
- **REQ_PS_PAGE_WRITE** : requête de copie en écriture.
- **REQ_PS_FIRST_TOUCH_READ** : requête de création de page en lecture.
- **REQ_PS_FIRST_TOUCH_WRITE** : requête de création de page en écriture.

10.3.6.1 Demande d'invalidation de page

Sur une demande d'invalidation, le serveur de pages appelle la fonction *invalidate_page*. Celle-ci supprime la page de la mémoire physique grâce à la fonction d'invalidation du lieur de bas niveau associé au conteneur (*ctnr_inv_page*). Puis, pour chaque liaison entre le conteneur et un service de haut niveau, le gestionnaire de page fait appel à la fonction *change_access* du lieur d'interface associé à la liaison pour invalider la page. Enfin, il invalide l'entrée correspondante dans la table des pages locale du conteneur et envoie un acquittement au nœud émetteur de la requête.

10.3.6.2 Demande de copie de page en lecture

Sur une demande de copie en lecture, pour chaque liaison entre le conteneur et un service de haut niveau, le gestionnaire de page fait appel à la fonction *change_access* du lieur d'interface associé à la liaison pour placer les droits de la page à lecture seule. Puis, il place les droits de la page dans la table des pages locale du conteneur à lecture seule, ajoute le nœud demandeur dans le *copyset* et envoie la page au nœud demandeur.

10.3.6.3 Demande de copie de page en écriture

Sur une demande de copie en écriture, le serveur de pages envoie des requêtes d'invalidation (*REQ_PS_PAGE_INVALID*) aux serveurs de pages des nœuds présents dans le *copyset* puis attend les acquittements en provenance de ces nœuds. Le serveur de pages envoie ensuite une copie de la page au nœud demandeur et invalide sa copie locale grâce à la fonction *invalidate_page*. Enfin, il invalide l'entrée correspondante dans la table des pages locale du conteneur.

10.3.6.4 Demande de création de page en lecture

Lors d'une demande de création de page en lecture, le serveur de page fait appel à la fonction *ctnr_first_touch* du lieur d'entrée/sortie associé au conteneur afin d'obtenir une copie de la page depuis le périphérique lié au conteneur. L'état de la page dans la structure de conteneur est placé à lecture seule, le nœud local ainsi que le nœud demandeur sont placés dans le *copyset*, enfin une copie de la page est envoyée au nœud demandeur.

10.3.6.5 Demande de création de page en écriture

Lors d'une demande de création de page en écriture, le serveur de page fait appel à la fonction *ctnr_first_touch* du lieur d'entrée/sortie associé au conteneur afin d'obtenir une copie de la page depuis le périphérique lié au conteneur. Une copie de la page est envoyée au nœud demandeur et la copie locale est détruite grâce à un appel à la fonction *ctnr_inv_page* du lieur d'entrée/sortie.

10.4 Mise en œuvre des lieurs d'interface

Le rôle des lieurs d'interface est de modifier l'interface des conteneurs pour permettre de les interfacer facilement avec les services systèmes du noyau hôte. En cela, leur mise en œuvre est extrêmement simple. La seule difficulté réside dans les mécanismes réalisant effectivement la connexion entre un conteneur et un service système grâce à un lieur d'interface. En effet, le seul changement d'interface des conteneurs ne permet pas de les lier directement à un service système. Quelques manipulations des structures de données de ce service et des structures de données du lieur utilisé sont nécessaires pour compléter le mécanisme de liaison.

10.4.1 Lieur d'interface de projection

L'interface de projection permet d'associer un conteneur à une VMA et ainsi de permettre d'accéder à ce conteneur via de simples lectures et écritures en mémoire virtuelle. Le lieur d'interface de projection remplace l'interface *get_page*, *grab_page* des conteneurs par une interface *no_page*, *wp_page* utilisée par le mécanisme de gestion de mémoire virtuelle.

Outre les fonctions *memory_no_page* et *memory_wp_page*, le lieur d'interface de projection est composé d'une variable nommée *mapping_linker*, d'une structure de donnée privée nommée *vm_linker_data* et de deux autres fonctions : *memory_change_access* et *link_vma_to_ctnr*. Le code associé à ces fonctions peut être trouvé en annexe F.1.1.

10.4.1.1 Liaison d'une VMA à un conteneur : la fonction *link_vma_to_ctnr*

La figure 10.9 présente la liaison d'une VMA à un conteneur. Un champs *vm_linker* a été ajouté à la structure *vm_area_struct* du noyau **[1]** afin de matérialiser la liaison entre une VMA et un conteneur. Ce champs pointe vers une structure *vm_linker_data* **[2]** qui représente les données du lieur associé à cette liaison. La structure contient un pointeur

vers le conteneur associé à la VMA, un pointeur vers la VMA et l'offset de cette projection dans le conteneur.

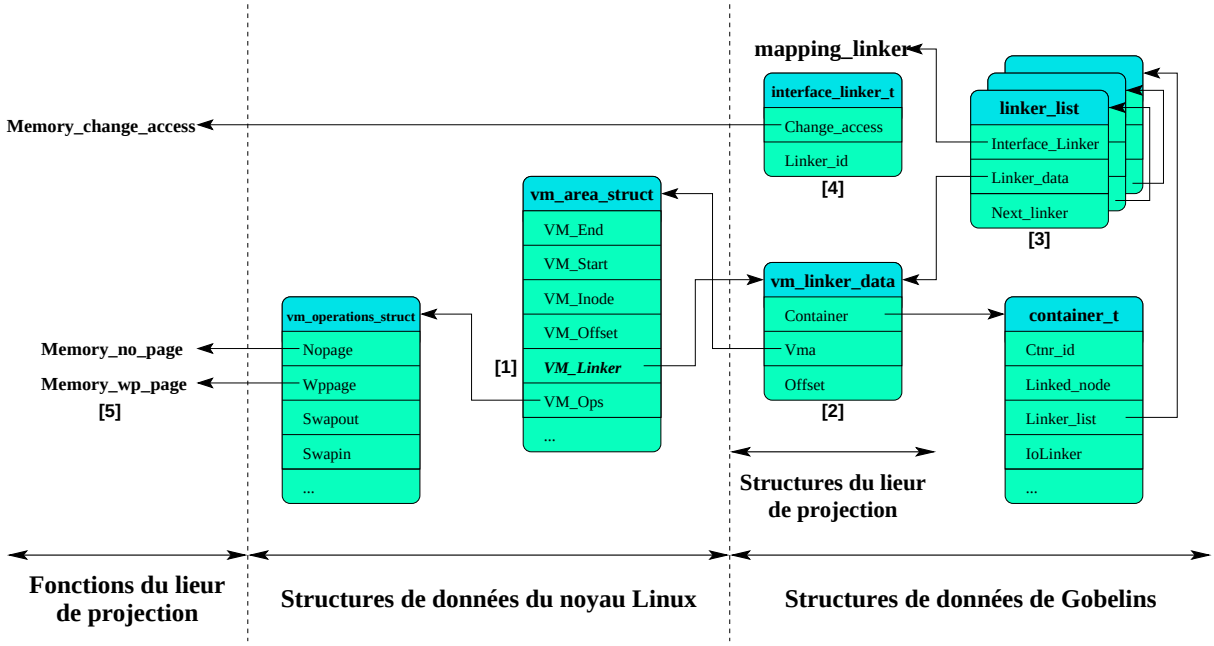

FIG. 10.9 – *Liaison d'une VMA à un conteneur*

Cette liaison est ajoutée à la liste des liaisons du conteneur grâce la structure *linker_list* **[3]**. La structure *linker_list* créée contient un pointeur vers la variable *mapping_linker* représentant le lieur d'interface de projection **[4]**. Cette variable de type *interface_linker_t* contient l'identifiant du lieur de projection et un pointeur vers la fonction *memory_change_access* mise en œuvre par le lieur. La structure *vm_linker* contient également un pointeur vers la structure *vm_linker_data* contenant les données privées de ce lieur pour la liaison considérée.

Enfin, les fonctions *no_page* et *wp_page* associées à la VMA **[5]** sont remplacées par les fonctions *memory_no_page* et *memory_wp_page* mises en œuvre par le lieur de projection. Une fois la liaison réalisée, tous les défauts de page intervenant dans la VMA sont traités par ces deux fonctions.

10.4.1.2 La fonction *memory_no_page*

La fonction *memory_no_page* commence par déterminer quel est le conteneur associé à la zone de mémoire en défaut de page grâce à un parcours de structures de données depuis le champs *vm_linker* de la structure *vm_area_struct* associée à la VMA.

La fonction *memory_no_page* peut être appelée suite à une lecture ou une écriture sur une page virtuelle qui n'est pas associée à une page physique. Dans le cas d'un accès en lecture, un appel à *ctnr_find_page* permet de déterminer si la page est présente localement.

Si c'est le cas, son compteur de référence est incrémenté et son adresse retournée. Si la page n'est pas présente, un appel à la fonction *ctnr_get_page* permet d'en obtenir une copie.

10.4.1.3 La fonction *memory_wp_page*

La fonction *memory_wp_page* détermine quel conteneur est associé à la VMA puis fait appel à la fonction *ctnr_grab_page* afin d'obtenir une copie en écriture de la page. L'adresse de la page est alors retournée.

10.4.1.4 La fonction *memory_change_access*

La fonction *memory_change_access* calcule l'adresse virtuelle de la page en fonction de son numéro dans le conteneur, de l'adresse de début de la VMA et de l'offset de liaison.

L'adresse de la page est passée en argument de la fonction *page_change_protection*. Celle-ci parcourt la page des pages du processus afin de trouver l'entrée correspondant à la page considérée. Les bits représentant ses droits d'accès sont alors mis en correspondance avec l'état de la page dans le conteneur, et l'entrée de la TLB correspondant à l'adresse de la page est invalidée.

10.4.2 Lieur d'interface fichier

Le lieur d'interface fichier n'a pas été intégralement mis en œuvre. Nous pouvons cependant présenter la mise en œuvre de celui-ci grâce au travail réalisé en vue de sa réalisation.

Le lieur d'interface fichier permet d'associer un conteneur à un fichier et ainsi de permettre d'accéder à ce conteneur via une interface de type « *read* »/« *write* ». Le lieur d'interface fichier remplace l'interface *get_page*, *grab_page* des conteneurs par une interface *find_page*, *write_page* utilisée par le gestionnaire de fichiers.

Outre les fonctions *file_find_page* et *file_write_page*, le lieur d'interface fichier est composé d'une variable nommée *file_int_linker*, d'une structure de donnée privée nommée *file_int_linker_data* et de deux autres fonctions : *file_change_access* et *link_file_to_ctnr*.

10.4.2.1 Liaison d'un fichier à un conteneur : la fonction *link_file_to_ctnr*

La figure 10.10 présente la liaison d'un fichier à un conteneur. Un champs *file_linker* a été ajouté à la structure *inode* du noyau **[1]** afin de matérialiser la liaison entre un fichier et un conteneur. Ce champs pointe vers une structure *file_int_linker_data* **[2]** qui représente les données du lieur associées à cette liaison. La structure contient un pointeur vers le conteneur associé au fichier, un pointeur vers l'i-nœud du fichier et la fonction originale d'écriture de page associée à l'i-noeud. Cette fonction est utilisée par le lieur pour réaliser les écritures effectives sur disque.

Cette liaison est ajoutée à la liste des liaisons du conteneur grâce la structure *linker_list* **[3]**. La structure *linker_list* créée contient un pointeur vers la variable *file_int_linker* représentant le lieur d'interface fichier **[4]**. Cette variable contient l'identifiant du lieur fichier

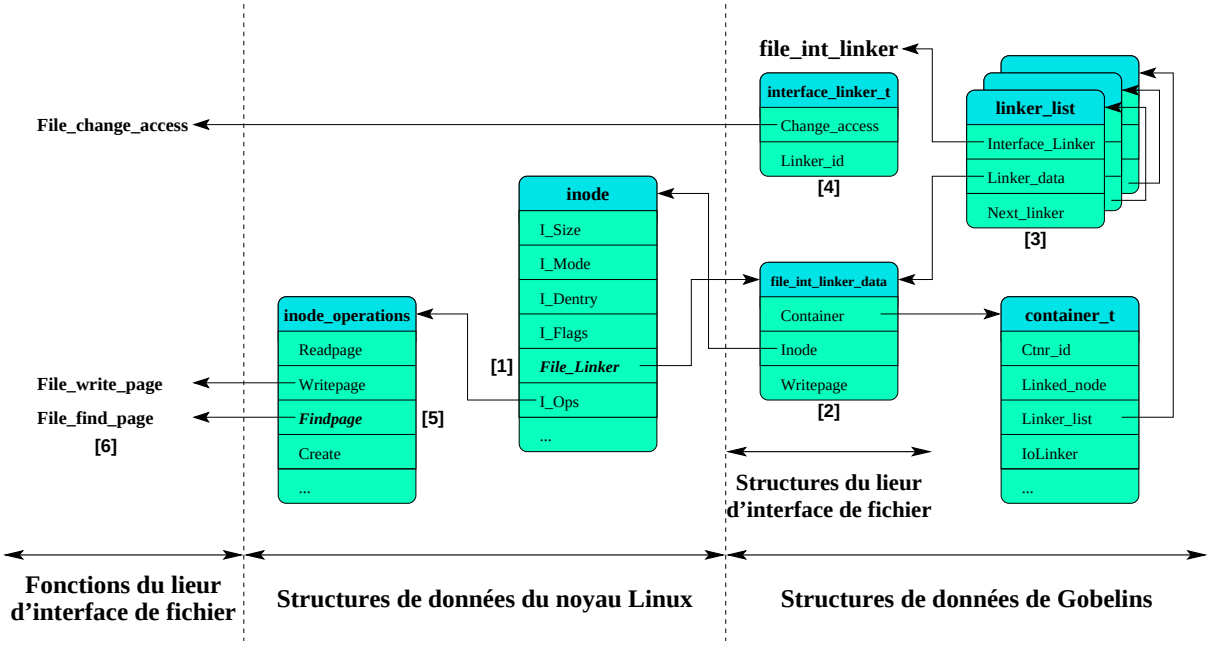

FIG. 10.10 – *Liaison d'un fichier à un conteneur*

et un pointeur vers la fonction *file_ change_ access* mise en œuvre par le lieur. La structure *file_ int_ linker* contient également un pointeur vers la structure *file_ int_ linker_ data* contenant les données privées de ce lieur pour la liaison considérée.

Enfin, les fonctions *find_ page* et *write_ page* associées à l'i-noeud **[6]** sont remplacées par les fonctions *file_ find_ page* et *file_ write_ page* mises en œuvre par le lieur d'entrée/sortie sur fichier. Une fois la liaison réalisée, tous les accès à ce fichier sont traités par ces deux fonctions.

Il est à noter qu'un champs a été ajouté à la structure *inode_ operations* **[5]** afin de pouvoir détourner la fonction *find_ page* du noyau vers la fonction du lieur de fichier. Ce champs n'est pas présent dans le code initial de LINUX. Le code du noyau qui faisait appel à la fonction *find_ page* a été modifié afin d'utiliser la fonction présente dans la structure *inode_ operations*. En l'absence de conteneur, c'est la fonction standard du noyau qui est appelée.

10.4.2.2 La fonction *file_ find_ page*

La fonction *file_ find_ page* commence par déterminer quel est le conteneur associé à la page du cache de fichiers demandée grâce à un parcours de structures de données depuis le champs *file_ linker* de la structure *inode* associée au fichier.

Si la fonction *file_ find_ page* est appelée pour une lecture de fichier, un appel à la fonction *ctnr_ find_ page* est effectué afin de déterminer si la page est présente localement. Si c'est le cas, son compteur de référence est incrémenté et son adresse retournée. Si la page

n'est pas présente, un appel à la fonction *ctnr_get_page* permet d'en obtenir une copie. Il est à noter qu'il n'est pas nécessaire de placer la page dans le cache de fichier. Cette opération est effectuée par le lieur d'entrée/sortie sur fichier.

Si la fonction *file_find_page* est appelée pour une écriture dans un fichier, la fonction *ctnr_grab_page* est appelée afin d'obtenir une copie unique de la page.

10.4.2.3 La fonction *file_write_page*

La fonction *file_write_page* parcourt les structures de données de liaison depuis le champs *file_linker* de la structure *inode* associée au fichier afin de déterminer quel est le conteneur associé. Le champs *linked_node* est consulté afin de connaître le nœud attaché au conteneur. Si ce nœud correspond au nœud local, la fonction fait appel à la fonction *write_page* initiale du l'i-nœud, grâce au champs *write_page* de la structure *file_int_linker_data*.

10.4.2.4 La fonction *file_change_access*

Comme nous l'avons présenté au paragraphe 9.1.2.6, la fonction *file_change_access* ne réalise aucune opération.

10.5 Mise en œuvre des lieurs d'entrée/sortie

10.5.1 Lieur d'entrée/sortie en mémoire

Le mécanisme de gestion de la mémoire physique de LINUX repose sur deux fonctions : *get_free_page* qui permet d'allouer un cadre de page et *free_page* qui permet de libérer un cadre de page. A chaque page physique est associé un compteur de référence permettant de connaître à tout instant le nombre de processus ou de services systèmes utilisant cette page.

Lorsqu'une page est allouée, ce compteur est placé à 1. Celui-ci peut être ensuite incrémenté par différents services systèmes comme par exemple le gestionnaire de mémoire virtuelle lorsqu'une page physique est projetée dans l'espace virtuel de plusieurs processus.

Lorsqu'une page est libérée, ce compteur est décrémenté. Lorsque ce compteur arrive à zéro, la page est placée dans la liste des pages libres.

La mise en œuvre du lieur d'entrée/sortie en mémoire repose exclusivement sur l'allocation et la libération de cadre de pages. Le code du lieur d'entrée/sortie en mémoire est présenté en annexe F.2.1.

10.5.1.1 La fonction *Memory_First_Touch*

La fonction *memory_first_touch* alloue un nouveau cadre de page en faisant appel à la fonction *get_free_page* du noyau LINUX et retourne son adresse.

10.5.1.2 La fonction *Memory_ Invalidate_ Page*

La fonction *memory_ invalidate_ page* localise dans la table des pages du conteneur, l'entrée correspondant à la page à invalider. L'adresse de la page physique est extraite de cette entrée et passée en argument de la fonction *free_ page* du noyau.

10.5.1.3 La fonction *Memory_ Free_ Page*

La fonction *memory_ free_ page* est identique à la fonction *memory_ invalidate_ page*.

10.5.1.4 La fonction *Memory_ Flush_ Page*

La fonction *memory_ flush_ page* n'a pas été mise en œuvre.

10.5.2 Lieur d'entrée/sortie sur fichier

Le code du lieur d'entrée/sortie sur fichier est présenté en annexe F.2.2

La gestion du cache de fichier est principalement assurée par six fonctions. Tout d'abord, la fonction *find_ page* permet de vérifier la présence d'une page dans le cache de fichier. Si la page est présente, son adresse est retournée. Il est à noter qu'un appel à cette fonction incrémente le compteur de référence de la page trouvée.

La fonction *add_ to_ page_ cache* permet d'ajouter une page dans le cache de fichier alors que la fonction *remove_ inode_ page* supprime une entrée du cache.

Les fonctions *page_ cache_ alloc* et *page_ cache_ release* permettent d'allouer et de libérer un cadre de page utilisé pour le cache de fichier.

Enfin, la fonction *wait_ on_ page* attend la fin d'un transfert depuis ou vers le périphérique associé à la page.

10.5.2.1 La fonction *File_ First_ Touch*

La fonction *file_ first_ touch* détermine quel est l'i-nœud du fichier associé au conteneur grâce au pointeur sur la structure de données du conteneur passée en argument. Puis un appel à la fonction *find_ page* permet de déterminer si la page est présente dans le cache de fichier du noyau.

Si la page n'est pas dans le cache, une entrée est allouée dans le cache grâce aux fonctions *page_ cache_ alloc* et *add_ to_ page_ cache*. Enfin, un appel à la fonction *read_ page* associée à l'i-nœud permet d'initialiser le chargement de la page.

Que la page ait été trouvée dans le cache ou chargée depuis le disque, un appel à la fonction *wait_ on_ page* permet de s'assurer que le chargement de la page est terminé. Une page située dans le cache peut très bien être en cours de chargement.

Enfin, l'adresse de la page est retournée.

10.5.2.2 La fonction *File_ Invalidate_ Page*

La fonction *file_ invalidate_ page* détermine quel est l'i-nœud du fichier associé au conteneur grâce au pointeur sur la structure de données du conteneur passée en argument. Puis un appel à la fonction *find_ page* permet de localiser l'entrée dans le cache de fichier hébergeant la page à invalider.

La fonction *wait_ on_ page* est appelée afin de s'assurer que tout transfert disque sur cette page est terminé. La page est libérée grâce à la fonction *page_ cache_ release* et supprimée du cache grâce à la fonction *remove_ inode_ page*.

Deux appels à la fonction *page_ cache_ release* sont nécessaires car la fonction *find_ page* incrémente le compteur de référence de la page. Un appel à la fonction *page_ cache_ release* permet d'annuler l'incrément réalisé par *find_ page* et un deuxième permet de réellement libérer la page.

10.5.2.3 La fonction *File_ Free_ Page*

La fonction *file_ free_ page* est identique à la fonction *file_ invalidate_ page*, à la différence que l'appel de cette fonction sur le nœud propriétaire provoque la recopie sur disque de la page considérée, si celle-ci a été modifiée. Si le nœud propriétaire n'est pas le nœud attaché, la page est envoyée au nœud propriétaire pour être recopiée sur disque grâce à la fonction *write_ page*.

10.5.2.4 La fonction *File_ Flush_ Page*

La fonction *file_ flush_ page* n'a pas été mise en œuvre.

10.6 Synthèse

La figure 10.11 présente le schéma de fonctionnement des conteneurs et des lieurs mémoire et fichier au sein du noyau LINUX. Nous présentons dans ce paragraphe les étapes de résolution d'un défaut de page dans une MPRL fondée sur l'utilisation de conteneurs ainsi que les étapes de lecture d'un fichier distant à travers les mécanismes de conteneur.

10.6.1 Résolution d'un défaut de page dans la MPRL de Gobelins

Lorsqu'un défaut de page se produit sur un nœud **[1]**, le gestionnaire de défaut de page de LINUX est activé et exécute la fonction *do_ page_ fault* **[2]**. L'exécution de cette fonction aboutit à l'appel de la fonction *memory_ no_ page* **[3]** du lieur d'interface mémoire. Si la page demandée n'est pas présente sur le nœud local, un appel à la fonction *ctnr_ get_ page* **[4]** est réalisé et aboutit à l'envoi d'une requête de copie en lecture au gestionnaire de la page **[5]**. S'il n'existe aucune copie de cette page dans la grappe et que la page appartient à un conteneur mémoire, un message de création de page est retourné au nœud demandeur **[6]**. Celui-ci fait alors appel au lieur d'entrée/sortie mémoire associé **[7]**, qui alloue un cadre de page grâce à la fonction dédiée du système LINUX **[8]**.

Dans le cas d'un conteneur fichier, le gestionnaire de page envoie une requête de création de page au nœud attaché **[6b]** qui appelle le lieur d'entrée/sortie fichier **[9]** et demande le chargement de la page au système de gestion de fichiers de LINUX **[10]**.

Enfin, s'il existe une copie de la page, le gestionnaire transmet la requête de demande de page au propriétaire **[11]**, qui retourne une copie de la page considérée au nœud demandeur.

10.6.2 Lecture d'un fichier distant dans GOBELINS

Lors d'un appel à la fonction de lecture sur fichier **[12]**, un appel système est réalisé afin de déclencher la fonction correspondante dans le noyau LINUX **[13]**. L'exécution de cette fonction aboutit à l'appel de la fonction *file_find_page* du lieur d'interface fichier **[14]**. Si la page demandée n'est pas présente sur le nœud local, un appel à la fonction *ctnr_get_page* **[4]** est réalisé et aboutit à l'envoi d'une requête de copie en lecture au gestionnaire de la page **[5]**. La suite du traitement est alors identique à ce qui a été présenté pour la résolution d'un défaut de page.

10.7 Résumé

Nous avons présenté dans ce paragraphe la mise en œuvre du système GOBELINS, qui est un système d'exploitation pour grappe fondé sur le concept de conteneur. La mise en œuvre des conteneurs et des lieurs a été intégralement réalisée au sein du système d'exploitation hôte LINUX. Seul le lieur d'interface fichier et le mécanisme d'injection n'ont pas été mis en œuvre. Leur mise en œuvre est cependant en cours et devrait aboutir dans les mois à venir.

La mise en œuvre du système GOBELINS est très légère, puisqu'elle ne représente que 3000 lignes de code, dont seulement 138 au sein même du code source de LINUX. Cependant, le code réalisé est techniquement très pointu et demande une très bonne connaissance de la programmation distribuée, du fonctionnement interne des systèmes d'exploitation, ainsi qu'une grande expertise de la programmation du noyau LINUX.

Nous avons choisi le système d'exploitation LINUX comme système hôte, mais la mise en œuvre des conteneurs peut être réalisée dans tout système d'exploitation utilisant la page mémoire comme unité de gestion des ressources physiques.

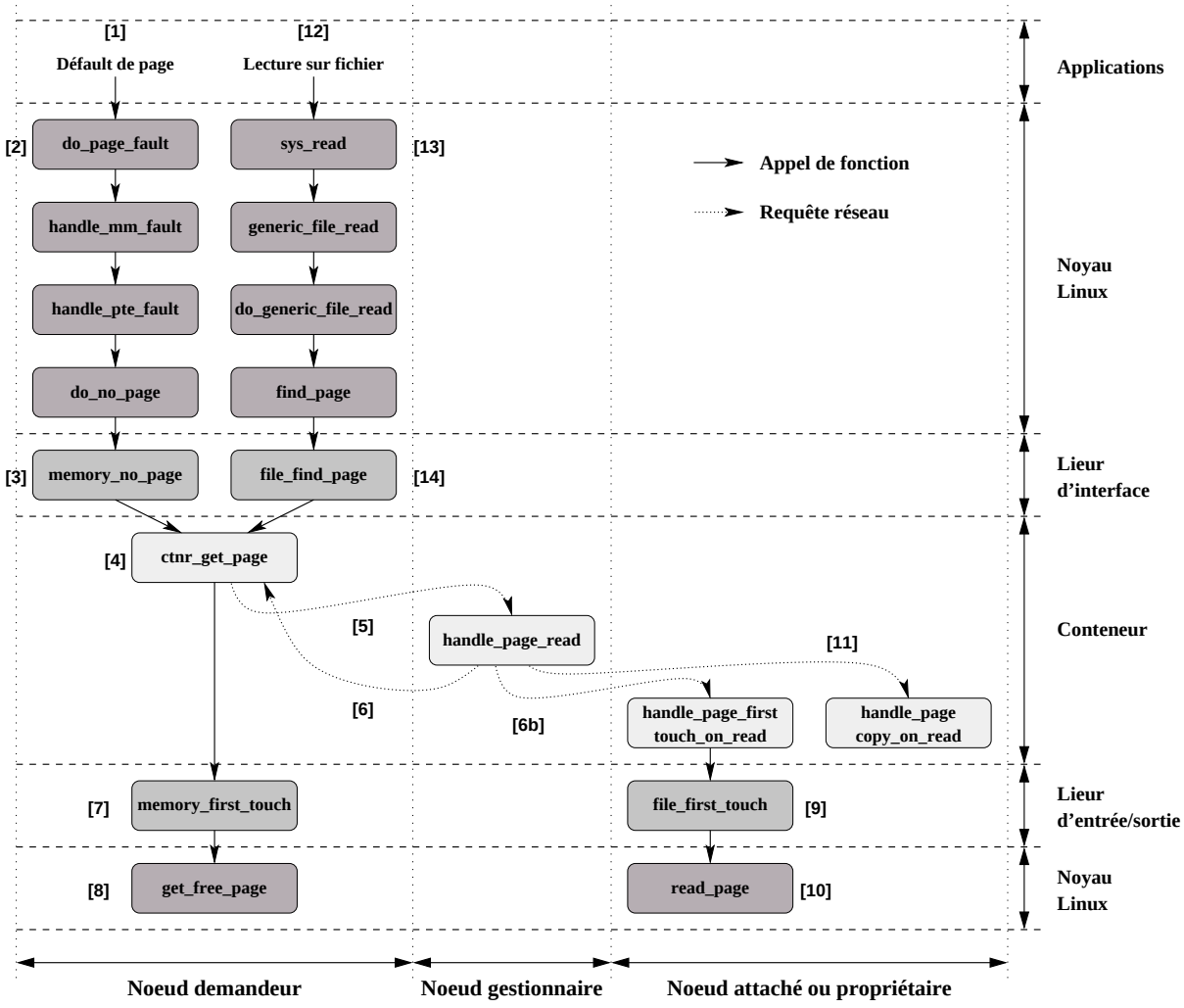

FIG. 10.11 – *Méthode de résolution d'un défaut de page et d'accès à un fichier via les mécanismes de conteneurs et de lieurs*

11 ÉVALUATION DE PERFORMANCES

Nous avons présenté au cours des chapitres précédents la conception et la mise en œuvre des conteneurs au sein du système d'exploitation GOBELINS. L'objectif des conteneurs est de permettre une mise en œuvre simple et efficace des mécanismes assurant la propriété de transparence pour les ressources mémoire, disque et processeur d'une grappe. Nous présentons dans ce chapitre une évaluation qualitative indiquant dans quelle mesure notre travail remplit les objectifs fixés. Nous présentons également une évaluation des performances de la mise en œuvre des conteneurs et des lieurs. L'évaluation porte sur plusieurs critères :

- l'évaluation des performances intrinsèques des conteneurs ;
- l'évaluation des performances d'une MPRL mise en œuvre grâce aux conteneurs ;
- l'évaluation des performances d'un cache de fichiers distribué coopératif mis en œuvre grâce aux conteneurs.

Nous détaillons dans la suite de ce chapitre les résultats de ces expérimentations et dégageons plusieurs pistes visant à améliorer ces performances et à combler les lacunes observées.

11.1 Conditions expérimentales

Ce paragraphe est destiné à présenter la plate-forme utilisée pour évaluer le prototype du système d'exploitation Gobelins, ainsi que l'état de ce prototype du point de vue de sa mise en œuvre.

En premier lieu, la plate-forme d'expérimentation est composée de 6 PCs interconnectés par un réseau Gigabit Ethernet. Les cartes réseaux sont connectées à un commutateur via des liens optiques. La configuration matérielle de chaque nœud est présentée dans le tableau 11.1. Chaque PC dispose de deux processeurs de type Pentium III, cependant en raison d'une limitation actuelle de la mise en œuvre de GOBELINS, un seul processeur est actif. Le support logiciel pour les machines multiprocesseurs n'a pas pu être réalisé à ce jour. Chaque nœud dispose également de 512 Mo de mémoire centrale et de deux disques durs SCSI, chacun d'une capacité de 9 Go. Les caractéristiques techniques des disques sont présentées dans le tableau 11.2.

L'état de la mise en œuvre de GOBELINS correspond à ce qui a été présenté dans le chapitre 10. La principale limitation de cette mise en œuvre concerne les mécanismes de remplacement décrit au paragraphe 8.5. Ceux-ci n'ont pu être mis en œuvre pour des raisons de temps de développement. Ainsi, il n'a pas été possible d'évaluer les performances de ce mécanisme et des mécanismes tirant directement partie de ces fonctionnalités comme par

Processeur	2 * Pentium III 500 MHz
Cache L2	512 Ko
Mémoire centrale	512 Mo
Disques	2 * IBM Ultra Star
Réseau	Packet Engine GNiic 2 (Gigabit Ethernet)

TAB. 11.1 – Configuration matérielle d'un nœud de la grappe d'expérimentation

Capacité	9.1 Go
Vitesse de rotation	10 020 RPM
Débit théorique	17 Mo/s
Cache interne	4 Mo
Latence	2.99 ms
Track to Track	0.7 ms
Nombre de plateaux	5
Nombre de têtes	10

TAB. 11.2 – Caractéristiques techniques des disques durs IBM DRVS-09V

exemple le cache de fichiers coopératif. En effet, une partie importante du fonctionnement d'un cache de fichiers coopératif consiste en cas de saturation de la mémoire locale à injecter des pages du cache vers les mémoires de nœuds distants.

D'autre part, le lieur d'interface fichier n'a pas été mis en œuvre. Tous les tests faisant intervenir des conteneurs fichiers ont été réalisés via une interface de projection de fichier en mémoire. Cependant, l'interface de fichier utilise les mêmes mécanismes que l'interface de projection. Nous pouvons ainsi estimer que les performances des accès aux fichiers via l'interface de fichier sont équivalentes à celles obtenues grâce à l'interface de projection. Enfin, nous avons désactivé les mécanismes de contrôle d'erreurs de la couche de communication de GOBELINS lors des tests de performance.

11.2 Évaluation qualitative

Nous présentons dans ce paragraphe une évaluation qualitative du système GOBELINS, indiquant dans quelle mesure les objectifs fixés au paragraphe 7.1 ont été remplis. Ces objectifs sont les suivants :

- (1) offrir l'image d'une machine unique ;
- (2) permettre l'exécution d'applications séquentielles et SMP sans aucune modification de leur code ;
- (3) tirer partie des performances potentielles de la grappe.

Le premier objectif concerne la mise en œuvre des propriétés énoncées au chapitre 2.3.2. Le second objectif est relatif à la transparence à l'utilisation du système à image unique

réalisé. Enfin, le troisième objectif est relatif à la performance du système réalisé. Nous traitons les deux premiers objectifs dans ce paragraphe. Le reste de ce chapitre étant consacré à l'évaluation du dernier objectif.

11.2.1 Transparence à la distribution

La version courante du système GOBELINS est la version *0.40*. Nous prévoyons une version *0.50* dans les mois à venir, contenant les mécanismes d'injection et de lieur d'interface fichier. Les propriétés offertes par ces versions sont présentées dans le tableau 11.3. La version *1.0* du système GOBELINS devrait intégrer toutes les propriétés énoncées au chapitre 2.3.2 permettant d'offrir un véritable système à image unique. L'intégration de travaux en cours concernant la gestion globale des processeurs et des disques complétera le travail présenté dans cette thèse.

	Gestion mémoire				Gestion disque				Cache		Gestion processeur			
	T.L.	I.L.	P.P.	P.L.	T.L.	I.L.	P.P.	P.L.	P.P.	P.L.	T.L.	I.L.	P.P.	P.L.
Gobelins 0.40	Oui	Oui	-	Oui	Inc[1]	Oui	-	Oui	-	Inc[2]	Oui	Inc[3]	Inc[4]	Oui
Gobelins 0.50	Oui	Oui	Oui	Oui	Inc	Oui	-	Oui	Oui	Oui	Oui	Inc	Oui	Oui
Gobelins 1.0	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui

Inc : incomplet.
[1] Un *thread* ne peut accéder à un fichier à distance que s'il a été ouvert par un autre *thread* s'exécutant sur le nœud hébergeant le fichier.
[2] Uniquement pour les fichiers projetés.
[3] Un processus migré ne peut pas communiquer avec ses fils et avec son père et ne peut pas utiliser de sockets.
[4] Le mécanisme de migration de processus a été mis en œuvre, mais n'est pas utilisé actuellement pour migrer automatiquement des processus ou *threads*.

TAB. 11.3 – Propriétés offertes par le système GOBELINS.

11.2.1.1 Gestion de la ressource mémoire

Les propriétés de transparence et d'indépendance à la localisation sont réalisées. L'accès à une donnée mémoire s'effectue de la même manière que celle-ci soit stockée localement ou à distance.

La propriété de partage logique est également réalisée puisqu'il est possible de partager des données mémoire entre des processus ou des *threads* s'exécutant sur des nœuds distincts.

Enfin, la propriété de partage physique n'a pas été réalisée à ce jour. Il est cependant prévu de la réaliser dans les mois à venir, grâce à mise en œuvre des mécanismes de remplacement.

11.2.1.2 Gestion de la ressource disque

La transparence à la localisation des fichiers est incomplète dans la version actuelle du système GOBELINS. Un *thread* ne peut accéder à un fichier à distance que s'il a été ouvert par un autre *thread* s'exécutant sur le nœud hébergeant le fichier. L'indépendance de localisation est cependant réalisée puisqu'un fichier distant est accédé de la même manière qu'un fichier local.

La propriété de partage logique est réalisée puisqu'il est possible à deux processus s'exécutant sur des nœuds distincts d'accéder simultanément à un même fichier.

Enfin, la propriété de partage physique n'est pas réalisée. Il n'est actuellement pas possible de créer un fichier stocké à distance grâce au système GOBELINS. Néanmoins, il est possible de le faire grâce aux outils standard du système LINUX (NFS, rcp, connexion à distance, etc). Cependant, ces solutions ne font pas partie du système GOBELINS et ne remplissent pas les critères de transparence que nous avons définis.

Ces propriétés seront assurées dans les versions futures du système GOBELINS grâce à la mise en œuvre d'un SGFD fondé sur les conteneurs tel qu'il a été décrit au paragraphe 9.3.4.

11.2.1.3 Gestion des caches de fichiers distribués

Le partage logique des caches de fichiers est assuré uniquement pour les fichiers accédés via l'interface de projection. Les fichiers accédés par l'interface de lecture/écriture ne profitent pas des données présentes dans les caches distants. De plus, la propriété de partage physique n'est pas assurée puisqu'un cache de fichiers ne peut pas placer de données dans un cache distant.

Ces propriétés seront assurées tout prochainement dans le système GOBELINS grâce à la mise en œuvre du lieur d'interface fichier et du mécanisme d'injection.

11.2.1.4 Gestion de la ressource processeur

Le travail présenté dans ce document ne vise pas directement à réaliser une gestion globale de la ressource processeur. Il offre néanmoins des mécanismes permettant de simplifier la mise en œuvre de mécanismes de migration de processus ou *threads* et d'améliorer les performances des processus ou *threads* migrés.

La transparence à la localisation des processus est assurée puisqu'il est possible d'exécuter un processus de la même manière et avec les mêmes résultats sur tous les nœuds de la grappe.

L'indépendance à la localisation est assurée de manière incomplète. Il est possible à un processus migré d'accéder aux données en mémoire et aux fichiers, qu'ils soient locaux ou distants. Cependant, un processus migré ne peut pas communiquer avec ses fils ou avec son père et ne peut pas utiliser de sockets. De plus, quelques appels systèmes ne peuvent pas être réalisés.

La propriété de partage logique est réalisée puisqu'il est possible d'exécuter des *threads* d'une même application sur des nœuds distincts de la grappe.

Enfin, la propriété de partage physique est réalisée de manière incomplète. Il est possible d'exécuter des *threads* d'une application sur des nœuds distants. Cependant, les processus et *threads* ne sont actuellement pas migrés automatiquement durant leur exécution. Cette propriété sera très prochainement intégrée dans le système GOBELINS. Des travaux sont en cours dans le cadre d'une autre thèse sur la conception d'un ordonnanceur global de processus dans le système GOBELINS [94].

11.2.2 Transparence à l'utilisation

Sur le système GOBELINS, il est possible d'exécuter des applications séquentielles et multithreadées. Les applications séquentielles s'exécutent sur les nœuds de la grappe de manière classique, sans aucune modification du code et sans aucun impact visible sur leur comportement. Dans la version actuelle du système GOBELINS, les applications s'exécutent sur le nœud où elles ont été lancées et ne sont pas migrées. Dans les versions futures du système, les applications séquentielles pourront être exécutées et migrées sur n'importe quel nœud de la grappe de manière transparence à l'utilisateur et sans aucune modification du code.

Les applications multithreadées peuvent s'exécuter de manière classique sur les nœuds de la grappe sans être modifiées. Les *threads* sont alors exécutées de manière classique sur le nœud où l'application a été lancée. Pour déployer ces *threads* sur les nœuds de la grappe et ainsi tirer partie des fonctionnalités de GOBELINS, il est nécessaire de modifier légèrement les applications et de les recompiler. Dans ce cas, les *threads* sont exécutés sur les différents nœuds de la grappe et partagent leur mémoire grâce à la MPRL de GOBELINS. La modification du code est mineure, puisqu'elle consiste à ajouter une fonction d'initialisation du mécanisme de placement de *threads* et à déplacer les allocations mémoire en début de programme. Il est en effet impossible actuellement d'allouer dynamiquement de la mémoire durant l'exécution d'une application SMP compilée pour GOBELINS.

Le second objectif n'est donc pas entièrement atteint puisqu'il est nécessaire de modifier une application SMP pour pouvoir l'exécuter sur l'ensemble des nœuds de la grappe. Cette limitation temporaire est liée à la mise en œuvre du mécanisme de migration de processus et à la gestion de l'allocation mémoire en MPRL. A l'heure actuelle, le mécanisme de migration est incomplet et ne permet pas de placer ou de migrer automatiquement des processus ou *threads* sans modification de leur code source. D'autre part, l'allocation dynamique de mémoire en MPRL grâce à la primitive *malloc* est délicate à mettre en œuvre et n'a pas pu être effectué dans le cadre de cette thèse. Ces limitations devraient néanmoins disparaître dans les mois à venir, nous permettant de remplir complètement l'objectif de transparence à l'utilisation et rendant possible l'exécution d'applications industrielles de grande envergure.

11.3 Performances intrinsèques

Nous avons réalisé un ensemble de mesures concernant le temps d'exécution des fonctions de base des conteneurs, à savoir *get_page* et *grab_page*. L'évaluation de ces fonctions a été menée pour les deux types de conteneurs : (1) les conteneurs mémoires et (2) les conteneurs fichiers.

Pour chaque série de tests, nous avons mesuré le temps nécessaire à l'exécution des fonctions *get_page* et *grab_page* dans les cas suivants :

- **cas de l'accès hors conflit** à une page dont il existe n copies dans la grappe, la valeur de n variant de 0 (premier accès) à *N-1*, N étant le nombre de nœuds de la grappe. L'accès hors conflit à une page signifie que lorsqu'un nœud accède à une page, aucun autre nœud ne réalise d'accès concurrent.

– **cas de l'accès simultané** par n nœuds à une page dont il n'existe qu'un seul exemplaire. La valeur de n variant de 2 à *N-1*.

Dans le cas d'un accès à une page hors conflit, aucun nœud n'effectue de requêtes concurrentes. Ainsi, les requêtes de demande de copie de pages sont traitées sans délai sur le nœud gestionnaire et sur le nœud propriétaire. De plus, il n'y a pas d'autre trafic sur le réseau que celui généré par la demande de copie de page du programme de test.

Pour chacune des fonctions, nous avons mesuré le temps d'exécution minimum, maximum et moyen lors de l'accès à 1024 pages consécutives. Les résultats présentés correspondent aux meilleurs temps mesurés sur une série de dix tests consécutifs.

Nous présentons dans les paragraphes suivants, les résultats obtenus pour chacune de ces expériences.

11.3.1 Conteneur mémoire

Ce paragraphe présente le temps d'exécution des fonctions *Get_Page* et *Grab_Page* lors de l'accès à un conteneur mémoire. Le tableau 11.4 présente les résultats des mesures de performance dans le cas d'un accès hors conflit à un conteneur mémoire.

Nb copies	Get_page			Grab_page		
	T_{min}	T_{moyen}	T_{max}	T_{min}	T_{moyen}	T_{max}
0	8	204	254	9	207	264
1	163	258	277	178	259	317
2	163	258	281	422	520	545
3	164	258	280	428	521	550
4	164	258	281	434	523	551
5	163	259	279	440	525	560

TAB. 11.4 – Temps d'exécution (en μs) des fonctions *Get_Page* et *Grab_Page* hors conflit d'accès

11.3.1.1 Coût d'une demande de copie de page hors conflit

La figure 11.1 présente les performances de la fonction *Get_Page*. Lors d'un accès à une page dont il n'existe aucune copie, le temps d'exécution minimum est de 8 μs et le temps d'exécution maximum de 254 μs. Dans ce cas, il n'y a pas de propriétaire, seul le gestionnaire intervient dans le protocole en retournant une requête de création de page au nœud effectuant le *Get_Page*. Le temps d'exécution minimum correspond au cas où le gestionnaire de la page demandé se situe sur le nœud faisant appel à la fonction *Get_Page*. Dans ce cas, aucune transaction réseau n'est réalisée. Le temps maximum correspond au cas où le gestionnaire est localisé sur un nœud distant. Dans ce cas, deux messages sont nécessaires à la résolution du protocole.

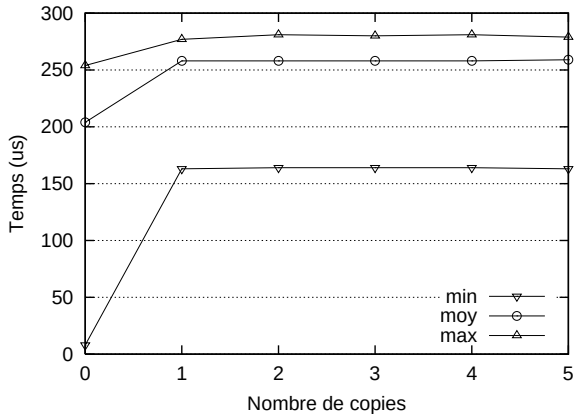

FIG. 11.1 – Temps d'exécution (en μs) de la fonction Get_page hors conflit sur conteneur mémoire

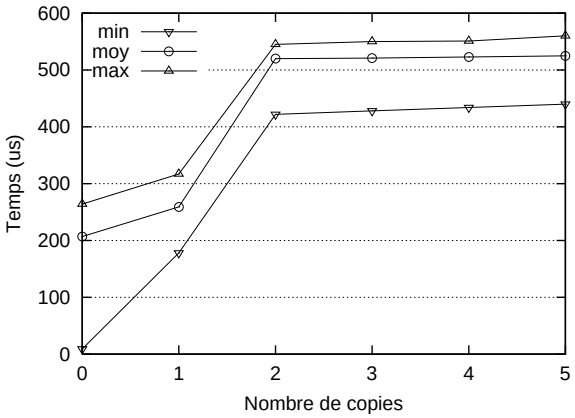

FIG. 11.2 – Temps d'exécution (en μs) de la fonction Grab_page hors conflit sur conteneur mémoire

Lors de l'accès à une page dont il existe une copie, le temps d'exécution minimum est de 163 μs, ce qui correspond au cas où le gestionnaire de la page est situé sur le nœud demandeur ou sur le nœud propriétaire. Dans ce cas, deux messages sont nécessaires à l'exécution du protocole, la requête du nœud demandeur vers le gestionnaire ou du gestionnaire vers le propriétaire étant effectuée localement. Le temps d'exécution maximum est de 277 μs. Il correspond au cas où le gestionnaire est situé sur un nœud distant. On a alors besoin de trois messages pour exécuter le protocole : (1) du nœud exécutant le *Get_ Page* vers le gestionnaire, (2) du gestionnaire vers le propriétaire et (3), du propriétaire vers le nœud demandeur.

On peut constater que dès lors qu'il existe au moins une copie d'une page dans la grappe, le temps d'exécution de la fonction *Get_ Page* reste constant quelque soit le nombre de ces copies.

11.3.1.2 Coût d'une demande de copie unique de page hors conflit

La figure 11.2 présente les performances de la fonction *Grab_ Page*. Lors d'un accès à une page dont il n'existe aucune copie, le temps d'exécution minimum est de 9 μs et le temps d'exécution maximum de 264 μs. Dans ce cas, il n'y a aucune copie et donc pas de propriétaire de page. L'exécution de la fonction *Grab_ Page* est alors équivalente à l'exécution de la fonction *Get_ Page*, puisqu'il s'agit de créer la première copie d'une page.

Lors de l'accès en copie unique à une page dont il existe déjà une copie, le temps d'exécution minimum est de 178 μs et le temps d'exécution maximum de 317 μs. Du point de vue des messages échangés et du temps d'exécution, ce cas est équivalent au cas d'un *Get_ Page* sur une page dont il n'existe qu'une copie, puisqu'il s'agit d'obtenir une copie de la page. A la différence du *Get_ Page*, la copie du nœud propriétaire est invalidée après son émission vers le nœud demandeur.

Enfin, lors de l'accès en copie unique à une page dont il existe plus d'une copie, le temps

d'exécution minimum varie de 422 à 440 μs et le temps d'exécution maximum varie de 545 à 560 μs. Dans ce cas, en plus de la copie du nœud propriétaire, il est nécessaire d'invalider les duplicatas. Le nœud propriétaire doit pour cela envoyer un message à chaque nœud disposant d'un duplicata et attendre un acquittement pour chacun d'eux. Pour chaque nœud disposant d'un duplicata, deux messages supplémentaires sont donc nécessaires. Or, on constate que le temps d'exécution est pratiquement constant quelque soit le nombre de copies (pour $n >= 2$). Ce phénomène s'explique par un recouvrement des demandes d'invalidation et des réceptions d'acquittement. L'algorithme utilisé envoie en rafale toutes les demandes d'invalidation, ce qui représente un temps d'exécution égal à n fois le temps d'initialisation d'une communication. Le premier acquittement est reçu au bout d'un temps équivalent à la transmission du message d'invalidation sur le réseau, son traitement par le nœud distant et la transmission d'un message d'acquittement en retour. L'acquittement suivant est reçu avec un retard équivalent au retard de la deuxième demande d'invalidation par rapport à la première, c'est-à-dire le temps d'initialisation d'une communication, et ainsi de suite. En recouvrant ainsi la transmission et le traitement des requêtes d'invalidation, le surcoût de l'invalidation se résume à quelques micro-secondes correspondant à l'initialisation d'une communication.

11.3.1.3 Coût d'une demande de copie de page avec conflit

La figure 11.3 présente les performances de la fonction *Get_Page* en présence de requêtes concurrentes. Le temps d'exécution minimum est obtenu pour une requête effectuée de manière concurrente par deux nœuds. On constate que le temps d'exécution minimum augmente assez peu avec le nombre de nœuds en concurrence. En revanche, le temps d'exécution moyen et le temps d'exécution maximum augmentent de manière significative avec le nombre de nœuds en concurrence : de 690 μs en présence de deux requêtes concurrentes à 893 μs en présence de cinq requêtes concurrentes pour le temps d'exécution maximum. Ces valeurs sont nettement supérieures à celles obtenues en l'absence de conflit.

Le temps minimum mesuré correspond au cas où le gestionnaire est situé sur le nœud demandeur et où la requête est traitée avec peu de délai par le propriétaire. En théorie, ce temps minimum devrait être identique au temps minimum observé hors conflit. Ceci correspondrait au cas exceptionnel pour lequel la requête du nœud demandeur est traitée sans aucun délai par le propriétaire. Ce cas semble suffisamment exceptionnel pour ne pas avoir été mesuré durant nos expérimentations.

Le temps maximum correspond au cas où le nœud demandeur, le nœud gestionnaire et le nœud propriétaire sont trois nœuds distincts. Dans ce cas, la requête à destination du gestionnaire peut être retardée d'un délai équivalent au temps de traitement d'une ou plusieurs requêtes par le gestionnaire. Ce délai est relativement faible puisqu'il correspond au temps de consultation d'une table et à l'émission d'un message de petite taille. La requête du gestionnaire vers le propriétaire peut être retardée d'un délai équivalent au traitement d'une ou plusieurs requêtes par le propriétaire. Ce délai est beaucoup plus long puisqu'il correspond à l'envoi d'une page sur le réseau.

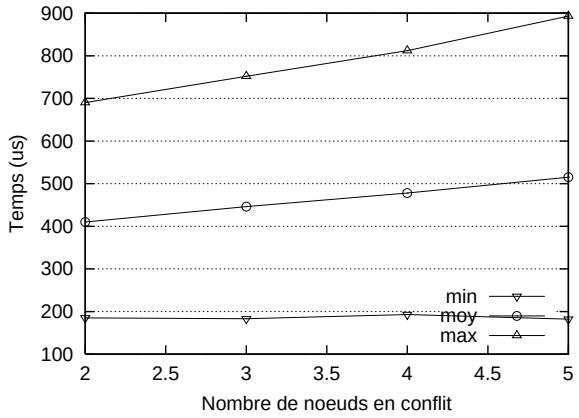

FIG. 11.3 – Temps d'exécution (en μs) de la fonction Get_Page avec conflit d'accès sur conteneur mémoire

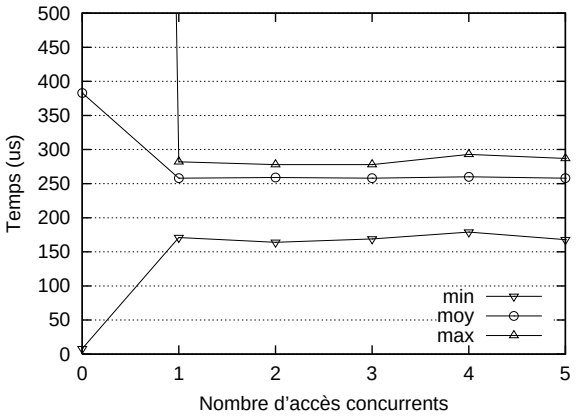

FIG. 11.4 – Temps d'exécution (en μs) de la fonction Get_Page hors conflit sur conteneur fichier

11.3.1.4 Optimisation

Le protocole de demande de copie de page est sérialisé à deux niveaux : (1) au niveau du gestionnaire de pages et (2) au niveau du propriétaire. Cette sérialisation a pour effet d'augmenter le temps de réponse de la fonctions *Get_page* en cas de conflit d'accès. La sérialisation au niveau du gestionnaire est peu pénalisante. En revanche, la sérialisation au niveau du propriétaire est beaucoup plus coûteuse du fait du traitement relativement long des requêtes qui lui sont destinées.

Le surcoût induit par un accès concurrent peut être réduit de manière significative en supprimant la sérialisation au niveau du propriétaire. Ceci est possible en exploitant la présence de duplicatas : lorsque le gestionnaire de pages reçoit une requête de copie en lecture sur une page, il pourrait avantageusement la transmettre à un nœud disposant d'un duplicata plutôt que de la transmettre systématiquement au propriétaire. Le gestionnaire peut choisir aléatoirement le nœud à qui transmettre la requête parmi la liste des nœuds disposant d'un duplicata. Statistiquement, on équilibre ainsi la charge d'émission des copies sur l'ensemble de ces nœuds.

La mise en œuvre de cette optimisation implique de modifier le protocole, notamment de déplacer ou de dupliquer le *copyset* au niveau du gestionnaire. La présence du *copyset* sur le nœud propriétaire est nécessaire pour la gestion du remplacement et pour des raisons de tolérance aux fautes non abordées dans ce document (voir [56] et [22]). La duplication du *copyset* semble donc préférable mais nécessite la mise à jour du *copyset* chez le propriétaire et donc l'émission d'un message supplémentaire. Une autre solution que nous n'avons pas étudiée consiste à disposer d'un *copyset* incomplet sur le nœud propriétaire et d'un *copyset* exact sur le nœud gestionnaire. Cette solution permettrait de conserver un équilibrage de la charge d'émission des copies sans surcoût lié à l'envoi d'un message supplémentaire. Cependant, les mécanismes de remplacement et de tolérance aux fautes seraient à revoir entièrement.

11.3.1.5 Coût d'une demande de copie unique de page avec conflit

Nous n'avons pas évalué le temps de traitement de la fonction *Grab_page* en cas de conflit. Nous considérons que ce cas représente une erreur de programmation de la part du concepteur de l'application. Ce cas correspond en effet à un faux partage induisant un phénomène de « *ping-pong* ». Ce phénomène est typique des protocoles de cohérence forte à invalidation sur écriture. Il ne peut être évité que grâce à une programmation soignée de la part du concepteur de l'application parallèle ou à l'utilisation d'un protocole à cohérence relâchée [55].

11.3.2 Conteneur fichier

Cette section présente les temps d'exécution mesurés pour la fonction *Get_Page* lors de l'accès à un conteneur fichier. Nous n'avons pas jugé nécessaire de mesurer le temps d'exécution de la fonction *Grab_Page* dans le cas des conteneurs fichiers. En effet, comme nous allons le voir dans la suite de ce paragraphe, la seule différence de performance observable entre un conteneur mémoire et un conteneur fichier est le coût d'un premier accès. Or, celui-ci est identique qu'il s'agisse d'un appel à la fonction *Get_page* ou à la fonction *Grab_page*.

Afin de mesurer l'impact de la localisation du nœud attaché à un conteneur fichier, nous avons effectué deux séries de mesures. La première consiste à accéder à des pages d'un conteneur attaché au nœud effectuant les accès, tandis que la deuxième série de mesures consiste à réaliser des accès à des pages d'un conteneur attaché à un nœud autre que le nœud effectuant les accès. Dans les deux cas, l'accès aux données est réalisé séquentiellement.

Le tableau 11.5 présente les résultats des mesures de performance dans le cas d'un accès hors conflit à un conteneur fichier attaché au nœud d'exécution et à un conteneur fichier attaché à un nœud distant.

Nb copies	Get_page local			Get_page distant		
	T_{min}	T_{moyen}	T_{max}	T_{min}	T_{moyen}	T_{max}
0	8	383	8552	165	535	9168
1	164	258	282	163	259	279
2	164	259	278	163	258	278
3	163	258	278	164	258	279
4	163	259	277	164	258	283
5	163	258	280	163	258	281

TAB. 11.5 – Temps d'exécution (en μs) de la fonction Get_Page pour un conteneur attaché localement et un conteneur attaché à distance

11.3.2.1 Coût d'une demande de copie de page hors conflit sur un conteneur attaché localement

La figure 11.4 présente les performances de la fonction *Get_ Page*. Lors d'un accès à une page dont il n'existe aucune copie, le temps d'exécution minimum est de 8 μs, le temps d'exécution maximum de 8552 μs et le temps d'exécution moyen de 383 μs. Le temps d'exécution minimum correspond au cas où le gestionnaire de la page est localisé sur le nœud demandeur et que la page demandée a été préalablement placée dans le cache de fichier par le mécanisme de préchargement. Le temps d'exécution maximum correspond au cas où la page doit être chargée depuis le disque et qu'il est nécessaire de déplacer le bras de lecture du disque afin de charger cette page. Ce temps d'exécution est très variable et dépend fortement de la topologie de stockage des données sur disque et de l'ordre des accès.

Lors d'un accès à une page dont il existe au moins une copie, les performances de la fonction *Get_page* sur un conteneur fichier sont identiques à celles observées sur un conteneur mémoire. A partir du moment où une page de données est chargée dans la mémoire de la grappe au sein d'un conteneur fichier, celui-ci se comporte exactement comme un conteneur mémoire à savoir qu'il permet de partager une page de données à travers les nœuds de la grappe. En effet, l'instanciation d'un conteneur consiste uniquement à modifier son comportement lors du premier accès et du remplacement d'une page.

11.3.2.2 Coût d'une demande de copie de page hors conflit sur un conteneur attaché à un nœud distant

Lors d'un accès à une page dont il n'existe aucune copie, le temps d'exécution minimum est de 165 μs, le temps d'exécution maximum de 9168 μs et le temps d'exécution moyen de 535 μs. Ces temps d'exécution correspondent à ceux observés dans le cas précédent augmentés du temps de transfert des données entre le nœud attaché et le nœud demandeur. Seul le temps d'exécution maximum n'est pas significatif. Du fait de la forte dépendance de cette valeur vis-à-vis de la localisation des données sur le disque, le temps d'exécution maximum peut varier de manière significative d'une exécution à l'autre.

11.4 Performance dans le cadre d'une MPRL

Nous avons évalué les performances des conteneurs lorsqu'ils sont utilisés pour réaliser une mémoire virtuelle partagée grâce à un ensemble d'applications parallèles programmées suivant un modèle de mémoire partagée et utilisant des *threads* Posix pour la gestion des activités parallèles. Trois applications ont été utilisées : *matmul*, *MGS* et *Jacobi*. Nous présentons dans la suite de ce paragraphe un rapide descriptif de ces applications et les performances obtenues lors des différentes séries de tests effectuées.

11.4.1 Description des applications de test

L'application *matmul* est un algorithme de multiplication de matrices carrées de taille n. Les deux matrices sources sont produites par un processeur et lues ensuite par tous les autres. Chaque processeur est alors chargé de calculer une partie de la matrice résultat. La distribution des calculs est fondée sur une distribution par bloc des lignes. A chaque processeur est associé un bloc de n/P vecteurs contigus, avec P le nombre de processeurs.

L'application *MGS* est un algorithme produisant une base orthonormée de l'espace généré par un ensemble de vecteurs indépendant, grâce à l'algorithme de Gram-Schmidt modifié. A chaque itération, un nouveau vecteur de la base orthonormée est calculé. A chaque fois qu'un processeur produit un vecteur, ce dernier est lu par chacun des autres processeurs. Ce vecteur est alors utilisé afin de corriger les vecteurs restants à normaliser dont chaque processeur a la charge. La distribution des calculs est réalisée grâce à une distribution cyclique des colonnes sur les différents processeurs. Chaque processeur i effectue les calculs intervenant sur les vecteurs v tels que v *modulo* $p = i$.

L'application *Jacobi* est un algorithme permettant de résoudre un système d'équations différentielles de Helmholtz grâce à la méthode itérative de Jacobi. A chaque itération de l'algorithme, la matrice calculée à l'itération précédente est recopiée dans une matrice temporaire. Cette matrice temporaire est alors utilisée pour calculer les éléments d'une nouvelle matrice. Chaque élément (i, j) de cette nouvelle matrice est calculé grâce à l'élément (i, j) de l'ancienne matrice et des quatre éléments voisins : $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ et $(i, j+1)$. La distribution des calculs est fondée sur une distribution par bloc des lignes. A chaque processeur est associé un bloc de n/p vecteurs contigus.

11.4.2 Étude de l'accélération

Nous avons tout d'abord évalué l'efficacité des conteneurs dans le cadre de la mise en œuvre d'une MPRL en mesurant l'accélération obtenue lors de l'exécution des applications de test sur notre grappe d'expérimentation. Le résultat de ces expérimentations est présenté dans la suite de ce paragraphe.

11.4.2.1 Conditions expérimentales

Pour chaque application, nous avons fait varier la taille de ensemble de données de 64*64 à 2048*2048 éléments flottants double précision afin d'évaluer l'impact de ce paramètre sur la performance. Nous avons également fait varier le nombre de nœuds utilisés de 2 à 6.

Pour ces trois applications, quelque soit la taille du problème, la matrice utilisée pour le calcul est stockée dans une matrice englobante de taille 2048*2048. Ceci permet d'éviter le faux partage en faisant coïncider les colonnes des matrices avec des frontières de pages mémoires [61].

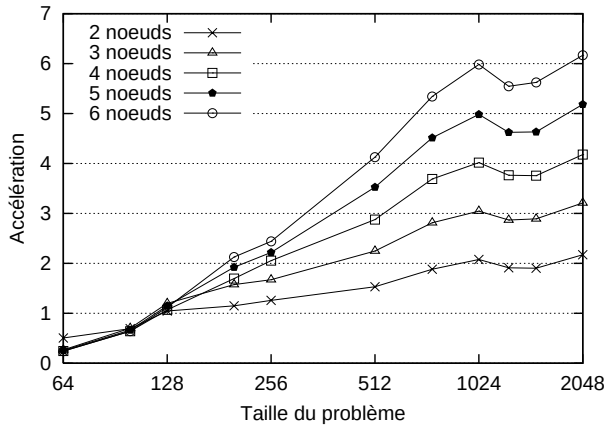

FIG. 11.5 – Accélération obtenue avec l'application de produit de matrices

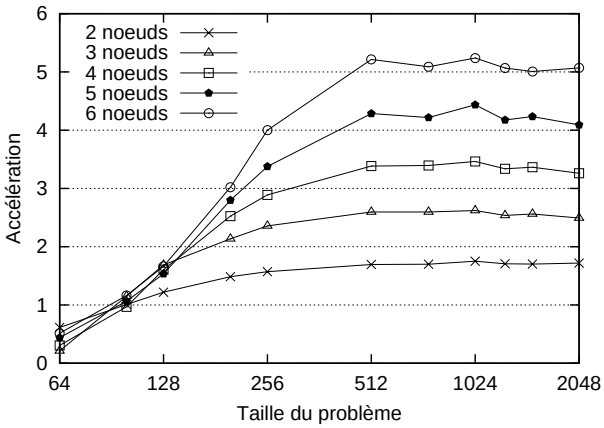

FIG. 11.6 – Accélération obtenue avec l'applications Jacobi

11.4.2.2 Résultats

Les figures 11.5, 11.6 et 11.7 présentent l'accélération obtenue avec les algorithmes de produit de matrice, de Jacobi et de Gram-Schmidt pour une taille de matrice variant de 64 à 2048 et l'utilisation de 2 à 6 nœuds. La figure 11.8 présente l'évolution de l'accélération des différentes applications en fonction du nombre de processeurs et pour une taille de problème de 2048*2048.

Pour l'algorithme de multiplication de matrices, on constate que l'accélération obtenue augmente lentement en fonction de la taille du problème jusqu'à atteindre l'accélération maximum théorique pour une matrice de taille 1024*1024. Au delà, l'accélération chute légèrement puis augmente à nouveau pour devenir super-linéraire sur une matrice de taille 2048*2048.

La chute de l'accélération entre les tailles 1024 et 2048 est due à une sous-utilisation des pages de mémoire virtuelle pour les tailles intermédiaires. En effet, si l'on considère par exemple une matrice de taille 1250*1250, chaque vecteur de cette matrice occupe en mémoire 1250*8 octets, ce qui représente 2,44 pages mémoire. L'accès à ce vecteur nécessite cependant de résoudre 3 défauts de pages. Or, moins de la moitié des données de la troisième page seront utilisées par l'algorithme, réduisant ainsi le rapport calcul/communication. Ce phénomène peut donc être observé pour toutes les tailles de problème non multiples de la taille d'une page mémoire. Cependant, l'impact de cette fragmentation interne a tendance à chuter avec l'augmentation de la taille du problème.

Pour l'algorithme de Jacobi, on constate que l'accélération obtenue augmente rapidement en fonction de la taille du problème, cependant, l'accélération n'atteint pas l'accélération maximum théorique. De plus, pour une taille de problème donnée, l'accélération obtenue s'éloigne rapidement de l'accélération optimale avec l'augmentation du nombre de nœuds. Cependant, à partir d'une taille de problème de 512, l'accélération est stable et l'on peut espérer conserver une accélération de bonne qualité avec l'augmentation de la taille

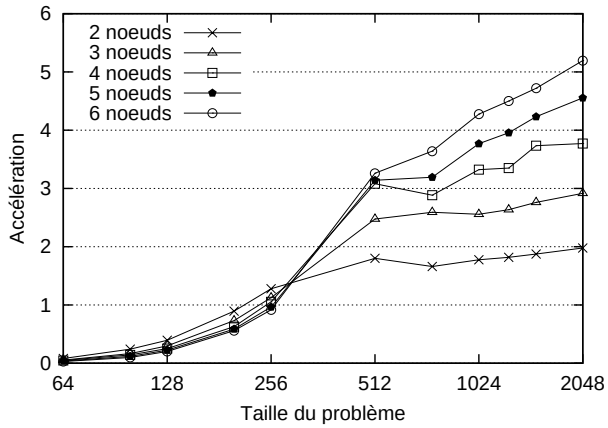

FIG. 11.7 – Accélération obtenue avec l'application Gram-Schmidt modifié

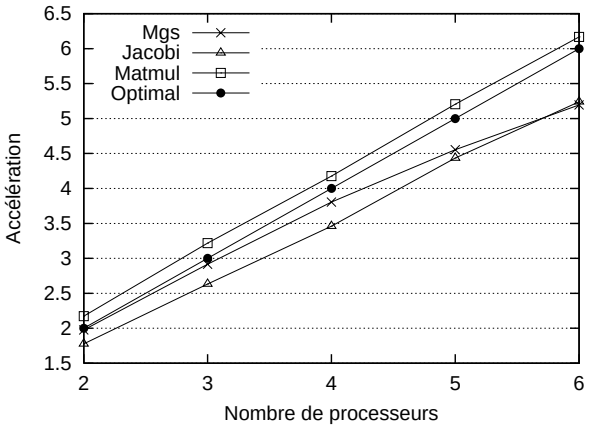

FIG. 11.8 – Évolution de l'accélération en fonction du nombre de nœuds

du problème et du nombre de processeurs.

Pour l'algorithme de Gram-Schmidt, on constate que l'accélération obtenue augmente lentement en fonction de la taille du problème, cependant, l'accélération n'atteint pas l'accélération maximum théorique. De plus, pour une taille de problème donnée, l'accélération obtenue s'éloigne rapidement de l'accélération optimale avec l'augmentation du nombre de nœuds (voir figure 11.8). L'accélération obtenue reste cependant de bonne qualité puisque pour un problème de taille 2048*2048, l'accélération sur 6 nœuds est de 5,2.

11.4.3 Comparaison de performances

L'objectif du système GOBELINS est d'offrir la vision d'une machine unique de type SMP au dessus d'une grappe, tout en essayant de tirer partie de la puissance potentielle de l'architecture matérielle sous-jacente. Le but est d'offrir une alternative aux calculateurs parallèles permettant d'obtenir de hautes performances à un coût financier très inférieur.

Afin d'évaluer le bien fondé de notre approche, nous avons comparé les performances du système GOBELINS s'exécutant sur une grappe de PCs à deux calculateurs SMP. Les résultats de ces expérimentations sont présentés dans la suite de ce paragraphe.

11.4.3.1 Conditions expérimentales

Nous avons comparé les performances obtenues sur trois architectures différentes lors de l'exécution des applications présentées dans le paragraphe précédent. Les trois architectures sont les suivantes : (1) une grappe de PCs exécutant le système GOBELINS, (2) un PC SMP quadri-processeurs et (3) un calculateur parallèle fondé sur une mémoire partagée de type cc-NUMA. Ces trois architectures disposent de processeurs de génération et de puissance équivalente. Leurs caractéristiques techniques sont présentées sur le tableau 11.6.

Les programmes de test utilisés sont programmés suivant un modèle de mémoire partagée en utilisant les *threads* POSIX pour créer et gérer les activités parallèles. Le même

	Grappe de PCs	SMP Dell 6300	Silicon Graphics Onyx 2
Processeurs	6 * Pentium III	4 * Xeon	6 * R12000
Fréquence	500 MHz	550 MHz	450 MHz
Data cache L1	16 Ko	16 Ko	32 Ko
Cache L2	512 Ko	512 Ko	4 Mo
Mémoire physique	512 Mo	1 Go	2 Go
Réseau	Gigabit Ethernet	Bus Mémoire	cc-NUMA
Système	GOBELINS	Linux 2.2.18	IRIX 64 6.5
Prix	160 KF	90 KF	1 MF

TAB. 11.6 – Caractéristiques techniques des calculateurs

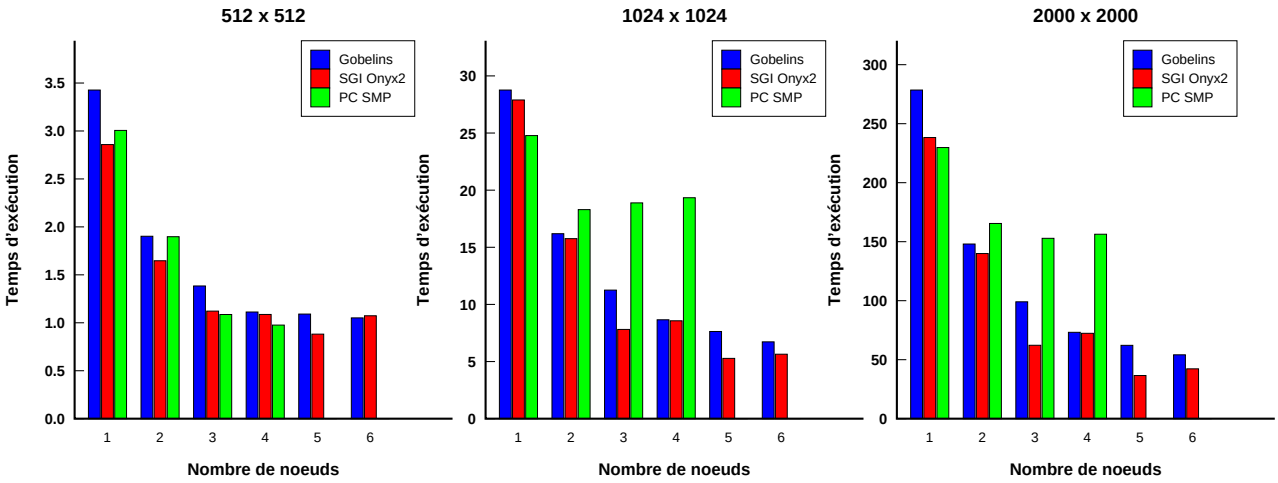

FIG. 11.9 – Temps d'exécution de l'algorithme de Gram-Schmidt Modifié

programme a été utilisé pour les trois architectures sans aucune modification de son code. Les programmes ont simplement été recompilés pour les différentes architectures.

11.4.3.2 Résultats

Les figures 11.9, 11.10 et 11.11 présentent les temps d'exécution obtenus pour les applications MGS, jacobi et multiplication de matrices pour les tailles de problème suivantes : 512*512, 1024*1024 et 2000*2000.

Les processeurs des différentes architectures sont de génération équivalente. On peut néanmoins constater que les temps d'exécution des différents programmes sont légèrement différents d'une architecture à l'autre dans le cas séquentiel (nombre de nœuds = 1), la machine Onyx 2 étant légèrement plus rapide. Cette légère différence de performance est à prendre en considération lors de la comparaison des performances dans le cas parallèle.

Pour l'application MGS, sur une taille de problème de 512*512 les trois architectures se

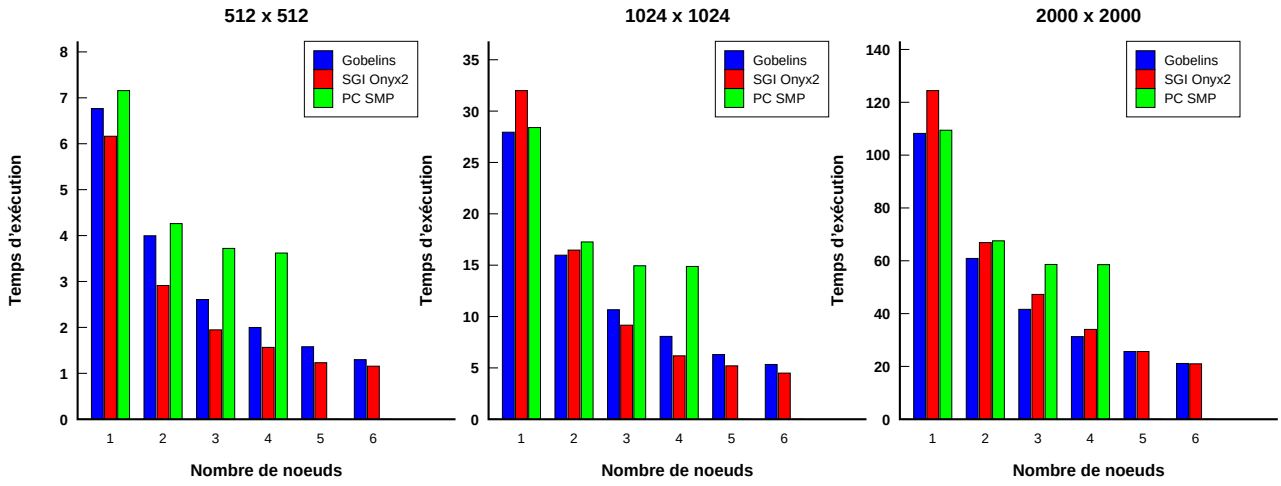

FIG. 11.10 – Temps d'exécution de l'algorithme de Jacobi

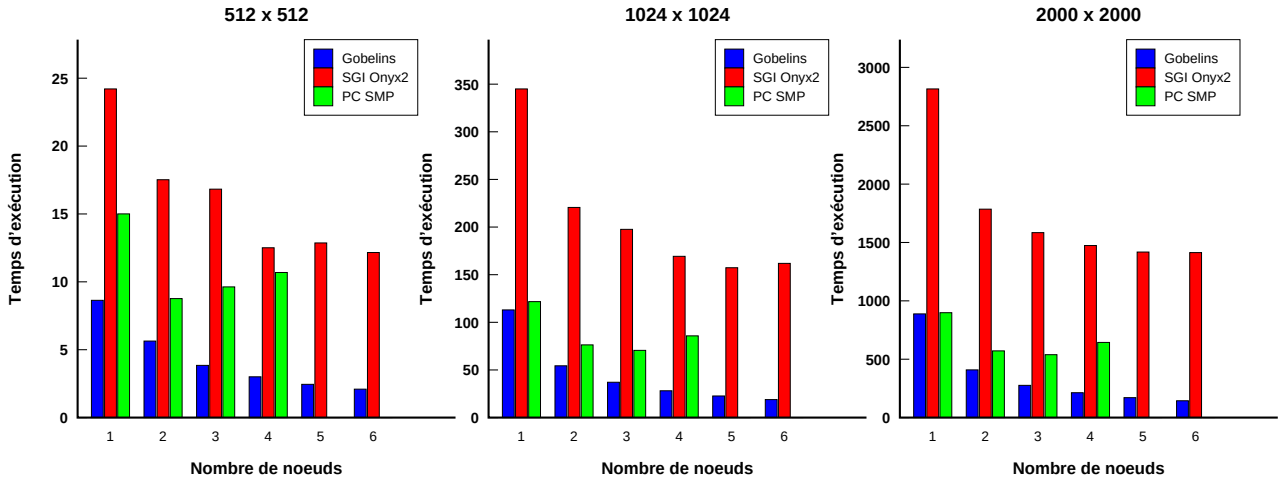

FIG. 11.11 – Temps d'exécution de l'application de multiplication de matrices

comportent sensiblement de la même manière : le temps d'exécution diminue proportionnellement à l'augmentation du nombre de processeurs et les différences de temps d'exécution sont minimes. Sur une taille de problème de 1024*1024 ou 2000*2000, GOBELINS continue à offrir de bonnes performances par rapport à la machine Onyx 2, alors que le PC SMP n'offre pas de réelle diminution du temps d'exécution avec l'augmentation du nombre de processeurs. Sur trois processeurs, les performances de GOBELINS sont sensiblement inférieures à celles offertes par la machine Onyx 2. Néanmoins, l'écart de performance se réduit avec l'augmentation du nombre de processeurs et de la taille du problème.

Pour l'application Jacobi, un comportement identique est observé. Cependant, quelque soit la taille du problème, le PC SMP n'offre pas de performances satisfaisantes. Ce phénomène est du à une insuffisance de la bande passante mémoire. Lorsque la taille du problème dépasse la taille du cache, le processeur produit de nombreux défauts de cache. Or, sur le PC SMP, lorsque les quatre processeurs effectuent simultanément des accès à la mémoire centrale pour remplir leurs caches, ils doivent se partager le débit mémoire. Ce débit étant insuffisant, les processeurs perdent de nombreux cycles de calcul à attendre le chargement des données à traiter. Ce phénomène ne se présente pas sur la grappe. Chaque nœud ne disposant que d'un seul processeur, lorsque celui-ci provoque de nombreux défauts de cache, il n'y a pas conflit d'accès à la mémoire et ainsi pas de contention provoquant le gaspillage de cycles de calcul.

Cette contention peut être évitée en modifiant les algorithmes afin de manipuler des sous-ensembles de données pouvant tenir en cache. Cependant, cette méthode augmente sensiblement la complexité des algorithmes et n'est pas toujours applicable. Par exemple, l'algorithme de Gram-Schmidt ne permet par un découpage du problème en sous-problèmes.

11.5 Performance dans le cadre d'un cache coopératif

Dans ce paragraphe, nous étudions les performances des conteneurs lorsqu'ils sont utilisés pour réaliser un cache de fichiers coopératif. Nous considérons le cas d'une application séquentielle s'exécutant sur un nœud de la grappe et réalisant des accès à un fichier. En l'absence du système GOBELINS, se pose le problème de la localisation du fichier. Dans ce cas, deux solutions sont possibles : (1) le stockage du fichier sur un serveur de fichiers et (2) le stockage du fichier sur le nœud local. Ces deux solutions ont des avantages et des inconvénients. Dans le cas d'un stockage sur le serveur de fichiers, le fichier peut être accédé facilement depuis n'importe quel nœud, cependant les performances d'accès seront médiocres. Dans le cas d'un stockage sur disque local, les performances d'accès seront beaucoup plus élevées, cependant le fichier ne pourra être accédé que sur ce nœud. Le système GOBELINS offre une nouvelle solution en permettant à tous les nœuds d'accéder à un fichier quelque soit sa localisation dans la grappe. Cette solution est proche d'une solution utilisant un serveur de fichiers, mais les performances sont accrues grâce à un mécanisme de cache de fichiers coopératif et à une mise en œuvre plus efficace.

11.5.1 Mesure du débit

Nous avons mené plusieurs séries d'expérimentations visant à comparer le débit d'accès à un fichier suivant la solution retenue pour son stockage, à savoir : (1) stockage sur disque local, (2) stockage sur un serveur de fichiers utilisant NFS et (3) stockage sur la grappe et accès via le cache coopératif de GOBELINS.

11.5.1.1 Conditions expérimentales

Quatre séries de mesures ont été effectuées dans les conditions suivantes :

- **Disque local** : le fichier est lu depuis le disque local grâce au SGF standard de LINUX (Ext2FS). Avant la lecture, le cache de fichiers du nœud d'exécution est froid. Ainsi, le fichier est entièrement lu depuis le disque.
- **NFS** : le fichier est lu à travers NFS sur un serveur distant situé en dehors de la grappe. Le cache de fichiers du nœud d'exécution est froid. Le cache de fichiers sur le serveur est chaud. Ainsi, le fichier est entièrement lu depuis le serveur, lequel ne réalise aucun accès disque.
- **Gobelins cold** : le fichier est lu à travers le cache coopératif de GOBELINS. Le cache de fichiers de LINUX ainsi que le cache coopératif de GOBELINS sont froids. Nous supposons que le nœud attaché est un nœud distinct du nœud d'exécution. Ainsi, le fichier est entièrement lu depuis le cache coopératif. De plus, celui-ci étant froid le fichier doit être chargé depuis un nœud distant dans le cache coopératif.
- **Gobelins hot** : le fichier est lu à travers le cache coopératif de GOBELINS. Le cache de fichiers de LINUX est froid alors que le cache coopératif de GOBELINS est chaud. Ainsi, le fichier est entièrement lu depuis le cache coopératif.

Chaque série de mesure a été effectuée dans le cas d'une lecture séquentielle et dans le cas d'une lecture aléatoire. La lecture aléatoire consiste à lire dans un fichier un volume de données correspondant au quart de la taille du fichier. Les données sont lues aléatoirement avec une granularité de 4 Ko. Une même donnée peut être lue plusieurs fois.

11.5.1.2 Résultats

Les figures 11.12 et 11.13 présentent les débits mesurés pour les quatre situations décrites précédemment dans le cas d'un accès en lecture séquentielle et dans le cas d'un accès en lecture aléatoire.

Dans le cas d'une lecture séquentielle, on constate que le débit en crête est de 14 Mo/s pour une lecture depuis le disque local, de 2 Mo/s pour une lecture sur NFS, de 7,5 Mo/s pour une lecture dans le cache coopératif froid et de 15,5 Mo/s pour une lecture dans le cache coopératif chaud. Dans le cas d'une lecture séquentielle, le comportement du cache coopératif est décevant puisque le gain de performance par rapport à une lecture sur disque local n'est que de 9.6% avec un cache coopératif chaud. Lorsque le cache coopératif est froid, l'accès à ce cache représente une perte de performance de 47.7% par rapport à l'accès au disque local.

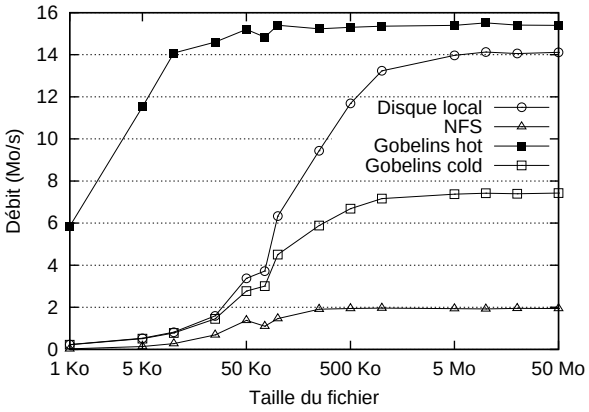

FIG. 11.12 – Débit en lecture séquentielle

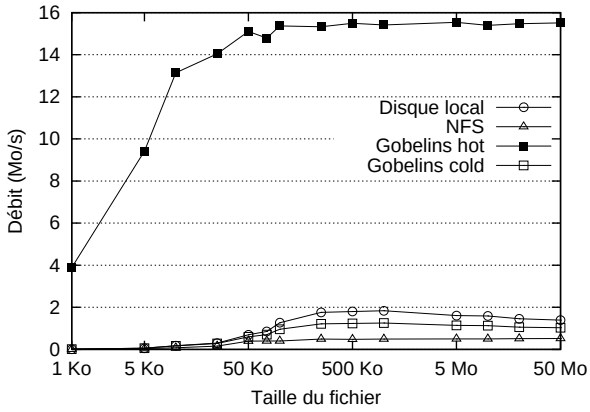

FIG. 11.13 – Débit en lecture aléatoire

Dans le cas d'une lecture aléatoire, on constate que le débit en crête est de 1,8 Mo/s pour une lecture depuis le disque local, de 0,5 Mo/s pour une lecture sur NFS, de 1,3 Mo/s pour une lecture dans le cache coopératif froid et de 15,5 Mo/s pour une lecture dans le cache coopératif chaud. On constate que le cache coopératif est totalement insensible à l'ordre des accès. La latence d'accès à une page du cache coopératif est constante quelque soit cette page. Dans le cas d'un disque dur, la latence de chargement d'une page dépend fortement de la localisation de cette page sur le disque par rapport à la position courante de la tête de lecture. Tout déplacement de la tête de lecture entraîne une augmentation significative de la latence. Dans le cas d'une lecture aléatoire, le cache coopératif se comporte très bien puisque le gain de performance par rapport à une lecture sur disque local est de 760% avec un cache coopératif chaud. Cependant, lorsque le cache coopératif est froid, l'accès à ce cache représente une perte de performance de 32% par rapport à l'accès au disque local.

11.5.2 Optimisation

11.5.2.1 Analyse des performances

Le comportement des conteneurs lorsqu'ils sont utilisés en temps que cache de fichiers coopératif est assez décevant. Le gain obtenu en lecture séquentielle n'est que de 9.6% alors que la dégradation de performance d'une lecture sur un cache coopératif froid est de 47.7% par rapport à une lecture locale.

Ces performances médiocres sont principalement dues à une sous-exploitation de la bande passante du réseau. Chaque défaut dans le cache de fichiers local d'un nœud provoque une requête de demande de page dans le cache coopératif. La page, présente ou non dans le cache coopératif est transmise par un nœud distant. Or, le débit obtenu sur un réseau Gigabit avec des messages de 4Ko n'est que de 30 Mo/s (voir annexe E.2). Si l'on ajoute à cela le temps de traitement du défaut de cache et de la gestion du protocole (localisation du propriétaire ou du nœud attaché), on constate que le débit final obtenu pour le cache de fichier coopératif n'est que 15,5 Mo/s.

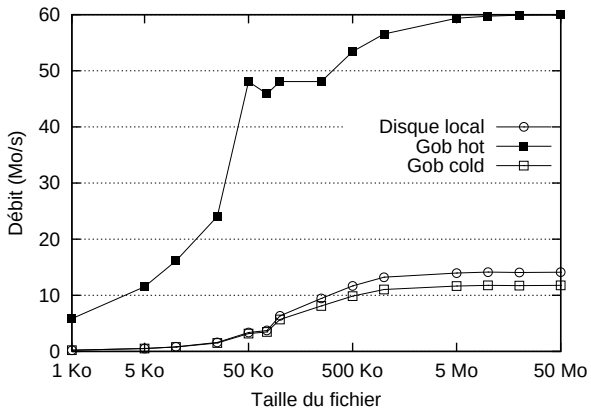

FIG. 11.14 – Débit obtenu en lecture séquentielle avec préchargement de 16 pages

11.5.2.2 Optimisation proposée

Une meilleure utilisation de la bande passante du réseau pourrait permettre d'augmenter le débit du cache coopératif. Une meilleure utilisation de la bande passante du réseau passe par l'utilisation de messages de plus grande taille.

Pour augmenter la taille des messages échangés lors d'un défaut dans le cache coopératif, on peut utiliser des mécanismes de préchargement. Lors du traitement d'une requête de copie de page sur le site propriétaire, le serveur de pages peut envoyer la page demandée ainsi que les *N-1* pages suivantes, *N* correspondant à la profondeur du préchargement. Ces N pages sont envoyées au sein du même message, augmentant ainsi la taille du message et donc le débit réseau. De plus, le coût de gestion du protocole n'intervient qu'une fois pour ces *N* pages au lieu de *N* fois dans le protocole actuel.

11.5.2.3 Résultats

Nous avons mis en œuvre l'optimisation proposée et évalué les performances pour $N = 16$. Les résultats obtenus pour un accès séquentiel sont présentés sur la figure 11.14. Le débit en crête est de 11,8 Mo/s pour une lecture dans le cache coopératif froid et de 60.9 Mo/s pour une lecture dans le cache coopératif chaud. On constate un gain de performance significatif par rapport à la version originale. Avec l'utilisation de préchargement, le gain de performance par rapport à une lecture sur disque local est de 331% avec un cache coopératif chaud, alors que la perte de performance lors d'un accès à un cache coopératif froid est désormais de 18,8%.

Afin d'évaluer l'impact de la valeur de *N* sur les performances, nous avons mené de nouvelles expérimentations en faisant varier cette valeur de 1 à 32. Les résultats sont présentés sur le tableau 11.7. Ce tableau présente pour chaque valeur de N, le débit en crête obtenu lors d'une lecture dans un cache coopératif chaud, le gain de performance obtenu par rapport à la lecture sur disque local et le surcoût induit par une lecture dans un cache coopératif froid par rapport à la lecture sur disque local. Le débit augmente fortement avec

l'augmentation de la profondeur de préchargement, pour atteindre près de 70 Mo/s pour une profondeur de 32. Dans le même temps, le surcoût induit par la lecture dans un cache froid chute jusqu'à atteindre 16,9% pour une profondeur de 32. Augmenter la profondeur de préchargement permet donc d'améliorer de manière spectaculaire les performances du cache de fichiers coopératif. Cependant, cette solution n'est pas indiquée dans le cas d'accès aléatoires puisque de nombreuses pages non utilisées risquent d'être transférées. Une étude plus poussée doit être effectuée afin de déterminer la valeur de N à utiliser en fonction du type d'accès effectués. On peut cependant remarquer que pour $N = 1$ dans le cas d'accès aléatoires, le cache coopératif offre une augmentation de performance de 760% par rapport à une lecture sur disque local.

	Valeur de N					
	1	2	4	8	16	32
Débit en crête	15.5	16.6	29.3	45.4	60.9	69.5
Gain cache chaud	9.6%	17.1%	107.7%	221.1%	331.1%	391.9%
Surcoût cache froid	47.7%	46.1%	32.5%	23.8%	18.8%	16.9%

TAB. 11.7 – Impact de la profondeur de préchargement sur les performances du cache coopératif

11.6 Conclusion

Nous avons présenté dans ce chapitre une évaluation qualitative du système GOBELINS ainsi qu'une évaluation des performances de services systèmes distribués mis en œuvre à l'aide de conteneurs dans le système GOBELINS.

L'évaluation qualitative du système GOBELINS indique que ce système rempli de manière satisfaisante les objectifs de transparence fixés dans ce travail, à savoir offrir l'image d'un système unique sur une grappe et permettre l'utilisation de ce système avec un minimum de modification des applications existantes. Les limitations existantes en terme de transparence sont dues au manque de temps pour finaliser la mise en œuvre et seront levées dans un futur proche.

Dans un deuxième temps, nous avons évalué les performances du système GOBELINS, notamment pour les services de mémoire virtuelle partagée et de cache de fichiers coopératif.

Dans le cas d'une mémoire virtuelle partagée, les résultats obtenus lors de l'exécution des applications de produit de matrices, de Jacobi et de Gram-Schmidt indiquent qu'il est possible d'obtenir une accélération élevée sur une grappe utilisant les conteneurs comme mécanisme de gestion d'une mémoire virtuelle partagée.

Nous avons cependant constaté qu'un calculateur parallèle offre des performances légèrement supérieures à celles offertes par le système GOBELINS, notamment pour des problèmes de petite taille. En effet, le surcoût induit par les mécanismes logiciels de partage de mémoire sont très pénalisant lors de l'exécution de problèmes de petite taille, le ratio

calcul/communication étant alors très faible. Dans le cas d'un calculateur parallèle, les mécanismes matériels de partage de mémoire ont un surcoût très inférieur à celui des mécanismes logiciels, permettant d'offrir un ratio calcul/communication élevé même pour des problèmes de petite taille. Cependant, les architectures à haute performance sont utilisées pour résoudre des problèmes de très grande taille, nécessitant des heures, voir des jours de traitement. Pour cette catégorie de problèmes, GOBELINS offre des performances comparables à celles obtenues sur un calculateur parallèle de puissance équivalente, ceci avec un coût financier d'un ordre de grandeur inférieur. Un effort de programmation supplémentaire est cependant à consentir pour éviter le phénomène de faux partage. Cet effort est équivalent à celui effectué sur les calculateurs parallèles afin d'optimiser l'utilisation du cache des processeurs.

Il faut néanmoins relativiser ces résultats au regard du nombre limité d'applications considérées et de la faible diversité des domaines d'application couverts. On peut cependant affirmer que pour la classe d'applications considérée, une grappe utilisant le système GOBELINS est une alternative réaliste aux calculateurs parallèles.

Dans le cas d'un cache de fichiers coopératif, les résultats obtenus indiquent qu'il est possible d'obtenir un débit très élevé lors de l'accès à un fichier situé dans un cache distant. Ce débit élevé représente un gain de performance très important par rapport à l'accès au disque local ou à un serveur de fichiers de type NFS. Ce gain de performance est d'autant plus important que l'accès au fichier est irrégulier.

12 CONCLUSION GÉNÉRALE

12.1 Bilan

Dans la littérature, on trouve de nombreux travaux visant à masquer la multiplicité des ressources d'une grappe. Les mécanismes de placement et de migration de processus permettent une gestion globale de la ressource processeur et assurent la transparence vis-à-vis de la localisation des processus sur les nœuds d'une grappe. Les mécanismes de mémoire virtuelle partagée offrent la vision d'une mémoire partagée unique au dessus d'un ensemble de mémoires distribuées assurant ainsi la transparence vis-à-vis de la localisation des données. Enfin, les mécanismes de gestion de fichiers distribués ou parallèles permettent le stockage et l'accès aux fichiers à travers l'ensemble des disques d'une grappe et assurent la transparence vis-à-vis de la localisation des fichiers.

Néanmoins, il n'existe à ce jour aucun système assurant une gestion globale de toutes les ressources d'une grappe, permettant d'offrir véritablement la vision d'une machine unique au dessus d'une grappe. La conception d'un véritable système à image unique passe par la gestion de toutes les ressources de la grappe et donc par l'utilisation de la plupart des mécanismes présentés. Cependant, chacun des systèmes proposés jusqu'à maintenant utilise des mécanismes spécifiques à chaque ressource, dont la mise en œuvre est très complexe. On peut cependant constater que la plupart des systèmes de gestion globale des ressources utilisent des mécanismes logiciels identiques : migration de page, gestion de la cohérence des données dupliquées, gestion de répertoires de localisation de données, etc. D'autre part, seuls les systèmes mis en œuvre dans le noyau d'un système d'exploitation offrent une parfaite transparence à l'utilisation et assurent toutes les propriétés de transparence pour une ressource donnée.

À la lumière de ces travaux, nous avons proposé une nouvelle architecture pour la conception d'un système distribué permettant de gérer l'ensemble des ressources d'une grappe. Pour cela, nous avons proposé de factoriser les composants logiciels utilisés par ces différents mécanismes de gestion globale des ressources. Cette factorisation a conduit à la proposition de la notion de conteneur et de lieur. Un conteneur est un objet logiciel permettant de stocker et d'échanger des pages entre les nœuds d'une grappe. Cette fonction de stockage et d'échange de données est en effet commune à tous les mécanismes distribués étudiés. Au sein d'un conteneur, la gestion des données dupliquées est assurée grâce à un mécanisme de cohérence forte. À nouveau, ce mécanisme de gestion de cohérence se retrouve dans de nombreux services distribués.

Un conteneur est intégré au sein d'un système d'exploitation hôte grâce à un ensemble de

lieurs. À chaque conteneur est associé une paire de lieurs : un lieur d'interface permettant de détourner les fonctions d'accès aux périphériques vers le conteneur et un lieur d'entrée/sortie permettant au conteneur d'accéder à un gestionnaire de périphérique donné. Pour chaque service système distribué de haut niveau, une paire de lieurs différente est utilisée. Par exemple, une paire de lieurs permettant de connecter un conteneur entre le gestionnaire de mémoire virtuelle et le gestionnaire de mémoire physique permet de réaliser une mémoire virtuelle partagée. Une paire de lieurs permettant de connecter un conteneur entre le système de gestion de fichiers virtuel et le gestionnaire de disque permet de réaliser un système de gestion de fichiers distribué et un cache coopératif.

Le mécanisme de conteneur allié à un ensemble de lieurs permet d'intégrer de nombreux services système distribuées au sein d'un système d'exploitation existant en le modifiant très peu. Il est également possible d'utiliser ce mécanisme comme base de l' architecture d'un nouveau système d'exploitation distribué écrit à partir de zéro. Un prototype nommé GOBELINS a été réalisé en intégrant les mécanismes de conteneur et de lieur au sein du système d'exploitation LINUX. Au sein de ce système, une mémoire virtuelle partagée, un cache de fichiers coopératifs et un mécanisme de projection distribuée de fichiers ont été mis en œuvre sur la base de conteneurs et de lieurs mémoire et fichier. La mise en œuvre des conteneurs représente 1700 lignes de code alors que la mise en œuvre des lieurs, et donc des services systèmes qui en découlent, ne représente que 1100 lignes de code. La modification du noyau lui-même reste extrêmement limitée, puisque moins de 200 lignes de code ont été modifiées ou ajoutées. Ceci démontre clairement la légèreté de conception et de mise en œuvre qui découle de l'utilisation des conteneurs au sein d'un système d'exploitation distribué.

Le système GOBELINS est actuellement utilisé au sein du projet de recherche Paris comme plate-forme d'expérimentation pour les travaux effectués dans le cadre d'autres thèses. Des applications scientifiques sont en cours de portage sur ce système et nous prévoyons également de porter des applications industrielles dans un avenir proche. Le portage de ces applications est extrêmement simple puisque le système GOBELINS offre les mêmes fonctionnalités qu'un système pour machine SMP et les modifications à apporter à ces applications sont très légères (ajout de quelques lignes de code et recompilation). Lorsque la mise au point de tous les mécanismes décrits dans cette thèse sera achevée, il sera possible d'exécuter des applications SMP sur le système GOBELINS sans aucune modification du code.

Les performances obtenues lors de l'exécution d'applications parallèles sur le système GOBELINS prouvent que les grappes représentent une alternative viable aux architectures parallèles en offrant un rapport performance/coût élevé. De plus, un système tel que le système GOBELINS permet de simplifier l'utilisation et la programmation des grappes et peut contribuer à la démocratisation de ce type d'architecture dans les PME/PMI. La généricité du concept de conteneur et la légèreté de la mise en œuvre nécessaire à son intégration dans un système existant permet d'envisager son utilisation dans d'autres systèmes d'exploitation employés dans l'industrie, tel que Solaris ou Windows NT.

12.2 Perspectives liées au prototype

Nous entrevoyons de nombreuses perspectives à ce travail. Les perspectives immédiates concernent le prototype réalisé :

- Une première perspective à court terme, est relative à l'amélioration du système GOBELINS afin d'y intégrer les mécanismes décrit dans ce travail et qui n'ont pu être mis en œuvre par manque de temps. C'est notamment le cas du lieur d'interface fichier et du mécanisme d'injection. L'intégration du mécanisme d'injection permettra notamment de tirer pleinement partie du mécanisme de cache coopératif. En effet, une partie importante du fonctionnement d'un cache de fichiers coopératif consiste en cas de saturation de la mémoire locale d'un nœud à injecter des pages du cache vers les mémoires de nœuds distants. Le mécanisme d'injection permettra également l'exécution d'applications gourmandes en terme d'utilisation mémoire sans remplacement sur disque. On peut espérer ainsi une augmentation significative des performances. Enfin, la mise en œuvre du lieur d'interface fichier permettra d'accéder à tous les fichiers stockés sur la grappe grâce à une interface de type lecture/écriture, tout en conservant les hautes performances du cache de fichiers coopératif.
- Il serait ensuite intéressant de valider le prototype grâce à un ensemble d'applications variées et représentatives des besoins industriels. Nous envisageons d'évaluer le système GOBELINS grâce à des applications de calcul scientifique, de simulation numérique et de serveur de données.
- Nous souhaitons également diffuser le système GOBELINS dans la communauté du logiciel libre et permettre son utilisation au sein de PME/PMI. Afin de satisfaire les besoins des entreprises, nous envisageons d'intégrer MPI et de concevoir un compilateur OpenMP pour GOBELINS.

12.3 Perspectives à long terme

Nous entrevoyons également plusieurs perspectives à plus long terme, dont les applications dépassent le cadre du système GOBELINS :

- Lors de l'évaluation de performances, nous avons mis en évidence l'efficacité des conteneurs. Cependant, l'accélération obtenue avec les différentes applications reste inférieure à l'accélération optimale théorique. De plus, nous avons mis en évidence l'impact du préchargement sur les performances du cache de fichiers coopératif sans pour autant définir quand et comment l'utiliser. La résolution de ces deux problèmes passe par une étude du comportement des applications afin d'adapter les politiques des conteneurs. Lorsqu'une application effectue des accès séquentiels à un fichier, il est souhaitable de précharger massivement les données depuis le cache coopératif, tandis que le préchargement doit être beaucoup plus raisonné dans le cas d'accès aléatoires. D'autre part, ce mécanisme de préchargement peut être utilisé avantageusement dans le cadre d'une mémoire virtuelle partagée lorsqu'un processus accède séquentiellement à des pages contiguës en mémoire. La réalisation de ces optimi-

sations passe par la conception d'un mécanisme permettant de caractériser le profil d'accès à un conteneur. Nous avons commencé l'étude de ce problème au travers d'un stage de DEA [93]. Ce travail est poursuivi dans le cadre d'une autre thèse. À partir d'un profil d'accès, il est possible de définir la profondeur de préchargement la mieux adaptée. Mais il est possible d'aller plus loin en essayant de déterminer des schémas d'accès caractéristiques comme des boucles, des accès par pas, des diffusions, etc, et d'utiliser ces informations pour prédire les futurs accès aux conteneurs. Il serait alors possible d'effectuer des préchargements, des diffusions matérielles mais aussi des optimisations dans le protocole de cohérence [62].

- Une autre approche pour améliorer la performance et simplifier la programmation est de relâcher la cohérence des duplicatas d'un conteneur. Relâcher la cohérence permet de résoudre les problèmes liés au faux partage, ce qui permet de limiter les échanges de données entre les nœuds et de libérer le programmeur de la contrainte liée à la gestion de ce faux partage.
- Nous avons présenté dans la première partie de cette thèse les quatre propriétés que doit remplir un système pour offrir la vision d'un système à image unique. Nous avons présenté dans cette thèse une solution permettant d'offrir la propriété de transparence à la distribution et à l'utilisation. Une autre propriété importante est la sûreté de fonctionnement. Les conteneurs représentant le point central de la conception d'un système distribué à image unique, l'introduction de mécanismes de tolérance aux fautes au sein des conteneurs semble indispensable pour assurer la propriété de sûreté de fonctionnement. Ces mécanismes sont également utilisés pour l'ajout et le retrait programmé de nœuds. Les mécanismes à introduire peuvent être séparés en deux catégories : (1) les mécanismes de haute disponibilité et (2) les mécanismes de reprise d'applications. Le premier mécanisme consiste à permettre aux conteneurs de fonctionner en présence de la défaillance d'un composant de la grappe. Actuellement, la gestion des conteneurs repose sur un ensemble de gestionnaires distribués sur les nœuds de la grappe. La perte d'un nœud implique la perte de ce gestionnaire et la défaillance du service de conteneur dans sa totalité. Des mécanismes de haute disponibilité peuvent être introduit afin de permettre au service de conteneur de poursuivre son fonctionnement après la défaillance d'un nœud. Nous avons travaillé dans cette voie au travers d'un stage de DEA [42]. Ces travaux sont poursuivis dans le cadre d'une autre thèse. D'autre part, la défaillance d'un nœud implique la perte des pages hébergées sur ce nœud et donc la défaillance de l'application utilisant ces pages. Des mécanismes de reprise d'applications peuvent être introduit dans la gestion des conteneurs afin de permettre la restauration d'une application après la défaillance d'un nœud. Nous avons participé à la conception d'un tel mécanisme au travers qu'une maquette inspirée du système Icare [56] et intégrant la gestion des données stockées sur disque [56] [22].
- Le mécanisme de conteneur a été créé afin de partager les ressources de type bloc à travers les nœuds de la grappe. Or, de nombreux périphériques de type caractère sont utilisés par un processus. L'absence de partage de ces ressources à l'échelle de la grappe est particulièrement problématique lors de la migration d'un processus.

Les solutions actuelles consistent à mettre en œuvre des mécanismes spécifiques à chaque ressource pour maintenir le lien entre le processus migré et le périphérique qu'il utilisait sur son nœud d'exécution initial. Une perspective de ce travail serait d'étudier la conception d'un mécanisme de partage des ressources de type caractère fondé sur une approche de type conteneur. Ce conteneur *caractère* permettrait d'offrir à l'utilisateur une vision parfaitement uniforme des ressources d'une grappe.

– Le mécanisme de conteneur permet actuellement de gérer les ressources d'une grappe. Cependant, l'architecture que nous avons proposée est peu extensible. La gestion distribuée des répertoires de pages et du protocole de cohérence nécessite l'envoi de messages de contrôle dont le nombre augmente sensiblement avec le nombre de nœuds. Ce problème peut être résolu grâce à une approche hiérarchique, en découpant une grappe en sous-grappes gérées grâce aux mécanismes que nous avons proposés. Nous n'avons cependant pas étudié les mécanismes permettant de fédérer les sous-grappes en une grappe vue comme une machine unique. Une autre perspective serait donc d'étudier les mécanismes nécessaires à la hiérarchisation d'une grappe.

BIBLIOGRAPHIE

[1] Small computer system interface (scsi) specification, 1986.

[2] A.Barak and A.Braverman. Memory ushering in a scalable computing cluster. In *Proc. IEEE Third International Conference on Algorithms and Architecture for Parallel Processing*, 1997.

[3] A.Barak and R.Wheeler. Mosix : An integrated multiprocessor unix. In *USENIX, winter*, 1989.

[4] A.Bricker, M.Litzkow, and M.Livny. Condor technical summary. Technical report, Computer Science Departement, University of Wisconsin, 1991.

[5] T. Anderson and R. Cornelius. High-performance switching with fiber channel. In *In Proceedings of IEEE CompCon*, pages 261–264, February 1992.

[6] T. Anderson, D. Culler, and D. Patterson. A case for now (networks of workstations). *IEEE Micro*, 15 :54–64, 1995.

[7] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1) :41–79, February 1996.

[8] Gabriel Antoniu and Luc Bougé. DSM-PM2 : A portable implementation platform for multithreaded DSM consistency protocols. In *Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'01)*, volume 2026 of *Lect. Notes in Comp. Science*, pages 55–70, San Francisco, April 2001.

[9] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River : Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, May 1999. ACM Press.

[10] Y. Artsy and R. Finkel. Designing a process migration facility : The charlotte experience. *IEEE Computer*, 22 :47–56, June 1989.

[11] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, and Tim Rühl. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1), February 1998.

[12] Brian N. Bershad and Matthew J. Zekauskas. Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report

CMU-CS-91-170, School of Computer Science, Carnegie-Mellon University, September 1991.

[13] R. Bhoedjang, J. Romein, and H. Bal. Optimizing distributed data structures using application-specific network interface software. In *Proceedings of the 1998 International Conference on Parallel Processing (ICPP '98)*, pages 485–492. IEEE USA, August 1998.

[14] Nanette J. Boden and al. Myrinet : A gigabit-per-second local area network. *IEEE Micro*, pages 29–36, February 1995.

[15] Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Madeleine : An efficient and portable communication interface for RPC-based multithreaded environments. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 240–247. IEEE Computer Society Press, October 1998.

[16] Peter Brezany and Alok Choudhary. Techniques and optimizations for developing irregular out-of-core applications on distributed-memory systems. Technical report, University of Vienna, November 1996.

[17] Rajkumar Buyya. *High Performance Cluster Computing*. Prentice Hall, 1999.

[18] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. I/O data mapping in *ParFiSys :* support for high-performance I/O in parallel and distributed systems. In *Euro-Par '96*, volume 1123 of *Lecture Notes in Computer Science*, pages 522–526, August 1996.

[19] J. Carretero, F. Pérez, P. de Miguel, F. García, and L. Alonso. ParFiSys : A parallel file system for MPP. *ACM Operating Systems Review*, 30 :74–80, April 1996.

[20] L.M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27 :1112–1118, December 1978.

[21] D. Cheriton. The v distributed system. *Communications of the ACM*, 31, March 1988.

[22] Renaud Lottiaux Christine Morin and Anne-Marie Kermarrec. High availability of the memory hierarchy in a cluster. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 134–143, October 2000.

[23] G. Ciaccio. Optimal communication performance on Fast Ethernet with GAMMA. *Lecture Notes in Computer Science*, 1388, 1998.

[24] D. Comer and J.Griffioen. A new design for destributed systems : The remote memory model. In *Proceedings of the Summer 1990 USENIX Conference*, pages 127–135, June 1990.

[25] Peter Corbett, Dror Feitselson, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traverstat, and Parkson Wong. MPI-IO : A parallel file I/O interface for MPI. Technical Report 19841 (87784), IBM T.J. Watson Research Center, Novemeber 1994.

[26] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3) :225–264, August 1996.

[27] T. Cortes, S. Girona, and J. Labarta. Avoiding the cache-coherence problem in a parallel distributed file system. In *High-Performance Computing and Networking*, volume 1225, pages 860–869, 1997.

[28] Toni Cortes. *Cooperative caching and prefetching in parallel/distributed file systems.* PhD thesis, Universitat Politècnica de Catalunya, 1997.

[29] Toni Cortes, Sergi Girona, and Jesùs Labarta. PACA : A cooperative file system cache for parallel machines. In *Proceedings of the 2nd International Euro-Par Conference*, pages I :477–486, August 1996.

[30] Mickael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching : Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating System Design and Implementation.* USENIX Association, November 1994.

[31] Vincent Danjean, Raymond Namyst, and Robert Russell. Integrating kernel activations in a multithreaded runtime system on Linux. In *Parallel and Distributed Processing. Proc. 4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, volume 1800 of *Lect. Notes in Comp. Science*, pages 1160–1167, May 2000.

[32] Alexandre Denis, Christian Pérez, and Thierry Priol. Towards high performance corba and mpi middlewares for grid computing. In *Proceedings of the 2nd International Workshop on Grid Computing*, November 2001.

[33] F. Douglis and J. Ousterhout. Transparent process migration : Design alternatives and the Sprite approach. *Software Practice and Experience*, 21 :1–27, July 1991.

[34] Cezary Dubnicki, Liviu Iftode, Edward W. Felten, and Kai Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[35] M. Dubois, C. Scheurich, and F.A. Briggs. Synchronization, coherence and event ordering in multiprocessors. *IEEE Computer Survey, Tutorial Series*, February 1988.

[36] E.Zayas. Attacking the process migration bottleneck. In *Proc. of the 11th ACM Symposium on Operating Systems Principles*, pages 13–22, November 1987.

[37] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing global memory management in a workstation cluster. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 201–212, December 1995.

[38] Michael J. Feeley. *Global Memory Management for Workstation Networks.* PhD thesis, Department of Computer Science and Engineering, University of Washington, December 1996.

[39] E.W. Felten and J. Zahorjan. Issues in implementation of a remote memory paging system. Technical report, Departement of Computer Science and Engineering, University of Washington, March 1991.

[40] Message Passing Interface Forum. MPI : A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[41] Roy Friedman, Maxim Goldin, Ayal Itzkovitz, and Assaf Schuster. MILLIPEDE : Easy parallel programming in available distributed environments. *Software Practice and Experience*, 27(8) :929–965, 1997.

[42] Pascal Gallard. Haute disponibilité dans les grappes de calculateurs. Master's thesis, Rapport de stage de DEA de l'université de Rennes 1, 2001.

[43] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. GLUnix : A Global Layer Unix for a network of workstations. *Software Practice and Experience*, 28 :929–961, July 1998.

[44] Richard B. Gillet. Memory Channel Network for PCI. *IEEE Micro*, 16 :12–18, February 1996.

[45] A.M. Goscinski, M.J. Hobbs, and J. Silock. Genesis : The operating system managing parallelism and provinding single system image on clusters. Technical report, Technical Repport TR C00/03, School of Computing and Mathematics, Deakin University, February 2000.

[46] David B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, pages 10–21, February 1992.

[47] Erik Hagersten, Anders Landin, and Seif Haridi. DDM — A cache-only memory architecture. *IEEE Computer*, 25(9) :44–54, September 1992.

[48] J. Hartman and J. Ousterhout. The zebra striped network file system. *ACM Transactions on Computer Systems*, August 1995.

[49] Robert L. Henderson. Job scheduling under the portable batch system. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 279–294. Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[50] J. V. Huber, C. L. Elford, D. A. Reed, and A. A. Chien. PPFS : A high performance portable parallel file system. In *Conference proceedings of the 1995 International Conference on Supercomputing*, pages 385–394. ACM Press, July 1995.

[51] Liviu Iftode, Cezary Dubnicki, Edward W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, February 1996.

[52] A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its applications in distributed shared memory systems. *The Journal of Systems and Software*, 47(1) :71–87, July 1998.

[53] Ayal Itzkovitz and Assaf Schuster. Distributed shared memory : Bridging the granularity gap. Technical Report CS-0953, Computer Science Department Technion, December 1998.

[54] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkinsk, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, Boston, MA, June 1985. IEEE.

[55] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks : Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[56] Anne-Marie Kermarrec. *Une approche globale fondée sur la réplication pour la disponibilité et l'efficacité des systèmes extensibles à mémoire partagée*. PhD thesis, Université de Rennes 1, 1996.

[57] Yousef A. Khalidi, Jose M. Bernabeu, Vlada Matena, Ken Shirriff, and Moti Thadan. Solaris MC : a multi computer OS. In *Proceedings of the USENIX Winter Technical Conference*, 1996.

[58] Povl T. Koch, J. S. Hansen, E. Cecchet, and X. Rousset de Pina. Scios : An sci-based software distributed shared memory. In *Proceedings of the 1st Workshop on Software Distributed Shared Memory*, pages 20–25, June 1999.

[59] Leonidas Kontothanassis, Galen Hunt, Robert Stets, Nikolaos Hardavellas, Michal Cierniak, Srinivasan Parthasarathy, Wagner Meira, Sandhya Dwarkadas, and Michael Scott. VM-based shared memory on low-latency, remote-memory-access networks. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2, pages 157–169, June 2–4 1997.

[60] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74, November 1994.

[61] Zakaria Lahjomri. *Conception et évaluation d'un mécanisme de mémoire virtuelle partagée sur une machine multiprocesseur à mémoire distribuée*. PhD thesis, Université de Rennes 1, January 1994.

[62] A. Lai and B. Falsafi. Memory sharing predictor : The key to a speculative coherent DSM. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, volume 27, 2, pages 172–185, May 1–5 1999.

[63] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transaction on Computers*, pages 690–691, September 1979.

[64] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, September 1986.

[65] W. B. Ligon and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 471–480. IEEE Computer Society Press, August 1996.

[66] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputer applications. In *Proceedings of Supercomputing '91*, pages 567–576, November 1991.

[67] M.Livny and M.Melman. Load balancing in homogeneous broadcast distributed systems. In *Proc. ACM Computer Network Performance Symposium*, 1982.

[68] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. Andrew : A distributed personal computing environment. *Communications of the ACM*, 29(3) :184–201, March 1986.

[69] Steven A. Moyer and Vaidy S. Sunderam. Parallel I/O as a parallel application. *The International Journal of Supercomputer Applications and High Performance Computing*, 9 :95–107, Summer 1995.

[70] F. Mueller. On the design and implementation of DSM-threads. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pages 315–324, June 1997.

[71] Raymond Namyst. *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.* Thèse de doctorat, Univ. de Lille 1, January 1997.

[72] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the SPRITE network file system. *ACM Transactions on Computer Systems, ; ACM CR 8903-0139*, 6(1), February 1988.

[73] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381. ACM Press, May 1996.

[74] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla Schlatter Ellis, and Michael Best. File-Access Characteristics of Parallel Scientific Workloads. Technical report, Dartmouth College, Computer Science, August 1995.

[75] Object Management Group. The Common Object Request Broker : Architecture and Specification (Revision 2.4), October 2000.

[76] John K. Ousterhout, A. R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, 21(2) :23–36, February 1988.

[77] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). Technical report, University of California, Berkeley, 1987.

[78] Elizabeth Pérez-Cortés and Jacques Mossiére. La cohérence sur mesure dans une mémoire partagée répartie. *Techniques et sciences informatiques*, 16(10) :1283–1310, December 1997.

[79] Eduardo Pinheiro and Ricardo Bianchini. Nomad : A scalable operating system for clusters of uni and multiprocessors. In *Proceedings of the 1st IEEE International Workshop on Cluster Computing*, December 1999.

[80] Eduardo Pinheiro, Ricardo Bianchini, Enrique Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, September 2001.

[81] G. Popek et al. The LOCUS distributed file system. *ACM SOSP-9, Bretton Woods NH*, October 1982.

[82] L. Prylli and B. Tourancheau. Bip : a new protocol designed for high performance networking on myrinet. In *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*, Lect. Notes in Comp. Science, pages 472–485. Springer-Verlag, apr 1998.

[83] R.Lottiaux and C.Morin. Containers : A sound basis for a true single system image. In *Proceeding of IEEE International Symposium on Cluster Computing and the Grid*, pages 66–73, May 2001.

[84] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network file system. In *Proceedings of Usenix 1985 Summer Conference*, pages 119–130, June 1985.

[85] M. Satyanarayanan. A study of file sizes and functionnal lifetime. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 96–108, December 1981.

[86] M. Satyanarayanan, J Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda : A highly available file system for a distributed workstation enrironment. *IEEE Transactions on Computers*, 39(4) :447–459, April 1990.

[87] D. J. Scales and K. Gharachorloo. Towards transparent and efficient software distributed shared memory. In *Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*, October 1997.

[88] Daniel J. Scales and Monica S. Lam. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI) : November 14–17, 1994, Monterey, California, USA*, pages 101–114, Berkeley, CA, USA, November 1994. USENIX.

[89] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *Proc. of the 6th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOSVI)*, pages 297–307, October 1994.

[90] Andrew Tanenbaum. *Systèmes d'exploitation*. Prentice Hall, 1992.

[91] K. Thitikamol and P. Keleher. Thread migration and communication minimization in DSM systems. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3) :487–497, March 1999.

[92] W. F. Tichy and Z. Ruan. Towards a distributed file system. In *Proceedings of the USENIX Summer Conference*, pages 87–97, Salt Lake City, Utah, June 1984.

[93] Gaël Utard. Prédiction des accès aux données pour l'amélioration des performances d'un système à mémoire partagée distribuée. Master's thesis, Rapport de stage de fin d'étude de l'Institut National des Sciences Appliquées (INSA), 2001.

[94] Geoffroy Vallée. Étude et mise en œuvre de la migration de processus au sein du système gobelins pour grappe de pcs. Technical report, Rapport technique EDF/IRISA/RESAM, 2001.

[95] Geoffrey M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 33–43, June 1998.

[96] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages : A mechanism for integrated communication and computation. In *Proceedings of the 19th annual International Symposium on Computer Architecture (ISCA'92)*, May 1992.

[97] V.Sunderam. Pvm : A framework for parallel distributed computing. *Concurrency : Practice and Experience*, pages 315–339, December 1990.

[98] Brent B. Welch. Measured performance of caching in the sprite network file system. *Computing Systems*, 1991.

Quatrième partie

Annexe

A RAPPEL DES PRINCIPES DES SYSTÈMES D'EXPLOITATION

A.1 Gestion de la mémoire

La mémoire physique d'un calculateur, aussi appelée mémoire vive ou RAM [1], est une zone de stockage volatile composée d'un ensemble de cases identifiées grâce à une **adresse physique**. Cette mémoire offre un accès rapide et un volume de stockage limité (de l'ordre de quelques centaines de méga-octets, à comparer aux dizaines de giga-octets offerts couramment par les disques durs modernes).

Définition 31 (Adresse physique) *On appelle* ***adresse physique*** *la référence à une case de la mémoire physique.*

Les programmes à exécuter sont chargés en mémoire physique depuis les unités de stockages secondaires (disques, CD-ROM, bandes, etc). Les données manipulées par les programmes sont également présentes en mémoire physique.

A.1.1 Mémoire virtuelle

La quantité de mémoire physique étant limitée pour des raisons de coût, le nombre de programmes chargeables simultanément et leur taille sont par conséquent eux aussi limités. D'autre part, dans un système d'exploitation multi-tâche et multi-utilisateur, il est fréquent d'observer la présence de plusieurs copies d'un même programme ou d'une même bibliothèque en mémoire physique, ce qui représente un gaspillage manifeste de cette ressource.

Définition 32 (Mémoire virtuelle) *La* ***mémoire virtuelle*** *est un mécanisme qui permet d'offrir au programmeur un espace d'adressage plus grand que la mémoire physique disponible.*

Un mécanisme de mémoire virtuelle remplit principalement quatre fonctions :
- offrir un espace d'adressage plus grand que la quantité de mémoire physique disponible ;
- regrouper des blocs de mémoire physique dispersés au sein d'une zone de mémoire virtuelle contiguë ;
- factoriser les multiples instances d'un même bloc de données ou d'instructions ;

[1] Random Access Memory

– offrir un mécanisme de contrôle d'accès aux données présentes en mémoire.

Ainsi, le mécanisme de mémoire virtuelle permet de charger et d'exécuter des programmes de taille plus importante que la mémoire physique et de pouvoir faire cohabiter au sein de la même mémoire un nombre plus important de processus.

A.1.2 Fonctionnement d'une mémoire virtuelle

Une mémoire virtuelle paginée est découpée en **pages** de taille fixe et la mémoire physique en **cadres de pages**. La taille de ces pages peut varier que quelques kilo octets (4096 octets sur un Intel Pentium, 8192 octets sur un alpha) à plusieurs méga octets (5 Mo sur un Pentium dans un mode particulier). Le processeur ne manipule pas directement des adresses physiques, mais des **adresses virtuelles**, qui sont transformées en adresses physiques par une **unité de gestion de mémoire** ou « *MMU* »[2].

Définition 33 (Cadre de page) *On appelle **cadre de page** une suite de N octets contiguës en mémoire **physique**, N étant une valeur fixée par la MMU.*

Définition 34 (Page) *On appelle **page** une suite de N octets contiguës en mémoire **virtuelle**, N étant une valeur fixée par le mécanisme de mémoire virtuelle.*

Définition 35 (Adresse virtuelle) *On appelle **adresse virtuelle** la référence à une case de la mémoire virtuelle.*

La gestion d'une mémoire virtuelle s'effectue à deux niveaux : au niveau matériel et au niveau système d'exploitation. Le matériel réalise la traduction des adresses virtuelles en adresses physiques et détecte les **fautes de page**, tandis que le système d'exploitation a la charge de résoudre les fautes de page.

Définition 36 (Faute de page) *Une **faute de page** ou **défaut de page** se produit lorsque le processeur référence une adresse virtuelle pour laquelle il n'existe aucune correspondance avec un cadre de page, ou lorsqu'une opération sur un mot mémoire ne respecte pas les droits d'accès associés à la page correspondante.*

Lorsqu'une adresse virtuelle est émise par le processeur, celle-ci transite par la MMU qui la découpe en deux parties : (1) un numéro de page et (2) un déplacement dans la page (c.f. figure A.1 **[1]**). Le numéro de page sert d'entrée dans une **table des pages [2]** qui contient les associations des pages virtuelles vers les cadres de page présents en mémoire physique. La combinaison du numéro de cadre de page issu de la table des pages et du déplacement dans la page produit une adresse physique **[3]** envoyée vers le module de mémoire physique.

A.1.3 Remplacement

Lorsque la quantité de mémoire physique disponible passe en dessous d'un seuil critique, le système d'exploitation active le mécanisme de **pagination**, ou **remplacement**

[2]Memory Management Unit

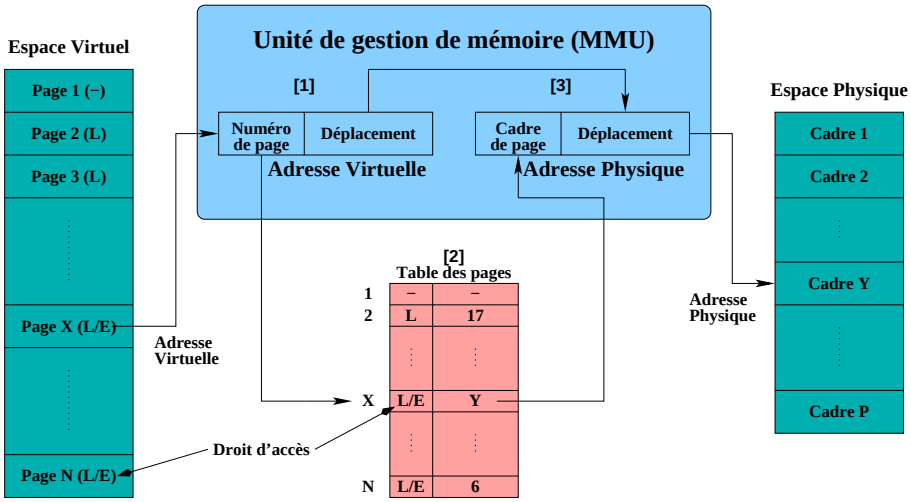

FIG. A.1 – *Mécanisme de mémoire virtuelle*

de pages. Ce mécanisme consiste à choisir des pages à évincer de la mémoire physique afin de libérer des cadres de pages. Les pages choisies sont transférées sur une unité de stockage secondaire, généralement un disque dur. Si une page évincée est référencée, une exception (faute de page) est levée pour réactiver le système d'exploitation qui charge la page en mémoire physique depuis l'unité de stockage secondaire.

Définition 37 (Pagination) *On appelle* ***pagination*** *le mécanisme, qui au sein d'une mémoire virtuelle assure l'échange de pages entre la mémoire physique et l'unité de stockage secondaire.*

Des heuristiques sont utilisées pour sélectionner les pages à évincer. L'objectif est de choisir les pages qui ont la probabilité la plus faible d'être utilisées dans un futur proche. La plupart des systèmes d'exploitation utilisent une approximation de LRU[3]. A chaque cadre de page est associée la date du dernier accès à celle-ci. Les pages les plus anciennement référencées sont choisies pour être évincées.

A.1.4 Protection mémoire

Les mécanismes de mémoire virtuelle sont également utilisés par les systèmes d'exploitation multi-utilisateur et multi-tâche pour réaliser l'isolation des espaces d'adressage des processus. Chaque processus dispose de son propre espace d'adressage à travers lequel il peut accéder à certaines pages physiques. Les pages physiques non projetées dans son espace d'adressage lui sont totalement inaccessibles. Par ailleurs, au sein d'un espace d'adressage, un **droit d'accès**, en lecture, écriture et exécution, peut être associé à chaque page de l'espace virtuel. Ce droit d'accès est stocké dans la table des pages et est consulté lorsque le processeur réalise une opération mémoire sur cette page. Si une opération provoque un

[3]Least Recently Used

accès non autorisé, par exemple la lecture d'un mot sur une page dont le droit en lecture est absent, le processeur lève une exception (faute de page) qui est gérée par le système d'exploitation.

A.2 Les systèmes de gestion de fichiers

Définition 38 (Système de gestion de fichiers) *Un **système de gestion de fichiers (SGF)** est un service système qui propose à l'utilisateur une interface d'accès simple et rapide aux unités de stockage secondaires (disques, bandes, etc.) grâce à l'abstraction de fichiers.*

Un SGF doit répondre à trois critères principaux : **abstraction** du matériel, **simplicité** d'utilisation et **efficacité**. L'abstraction du matériel est assurée grâce à la notion de **fichier**. Un fichier est une suite d'octets contiguës stockés sur disque. Il est possible de charger tout ou partie d'un fichier en mémoire physique grâce à une **interface d'accès**. Pour assurer l'efficacité des accès, il est nécessaire de masquer la latence élevée et la faible bande passante des disques. Pour cela, des mécanismes de **cache** et de **pré-chargement** sont utilisés.

A.2.1 Interface d'accès aux fichiers

On peut classer les opérations effectuées par l'interface d'accès aux fichiers d'un SGF en trois catégories : la **désignation** des fichiers au moyen de noms symboliques, la **protection** des fichiers contre des utilisations abusives et bien entendu, l'**accès** au contenu des fichiers [90].

A.2.1.1 Désignation

L'espace de désignation est formé par l'ensemble des fichiers stockés sur le disque qui sont de deux types : les **fichiers standard** qui contiennent les données des utilisateurs et les **répertoires** qui référencent des fichiers pouvant éventuellement être également des répertoires.

Chaque fichier possède deux types de noms : des **noms symboliques** ou **externes** et un **nom interne** unique. Le nom externe est exploité par l'utilisateur comme identificateur et voie d'accès au fichier. Il consiste en une chaîne de caractères attribuée par l'utilisateur. Le nom interne est utilisé par le SGF pour localiser le fichier sur le disque. La correspondance entre un nom externe et le nom interne est enregistrée dans un répertoire lors de la création du fichier et supprimée lors de sa destruction. Chaque fichier est globalement identifié par un **chemin d'accès** depuis la racine jusqu'au fichier, aussi appelé **nom absolu**.

A.2.1.2 Protection

Étant donné le partage de l'arborescence entre tous les utilisateurs, l'interface d'un SGF doit mettre en œuvre un mécanisme de protection des fichiers afin de spécifier les droits d'accès sur les fichiers (lecture, écriture, exécution) pour une catégorie d'utilisateurs

et associer un propriétaire au fichier. Le nom du propriétaire du fichier et la liste des droits d'accès constituent les attributs nécessaires au traitement de la protection.

A.2.1.3 Accès logique

Les fichiers standard, accédés au sein d'une session (délimitée par les opérations d'ouverture et de fermeture), sont vus logiquement comme un tableau infini d'octets. Ce tableau peut être consulté ou modifié à partir de n'importe quel indice, pour un nombre quelconque d'octets et peut changer de taille dynamiquement en le tronquant ou l'étendant. Pour mettre en œuvre ces opérations, il est nécessaire de conserver la taille et la **position courante** dans le fichier. Celle-ci indique la position dans le fichier où aura lieu la prochaine opération. L'accès logique aux fichiers est assuré grâce à une interface composée principalement de 5 fonctions :

- **ouverture** : permet d'initialiser l'accès à un fichier et d'associer un nom interne au nom externe fourni par l'utilisateur.
- **fermeture** : termine l'accès à un fichier, libère les structures de données associées.
- **lecture** : permet de charger en mémoire physique des données stockées dans la fichier.
- **écriture** : permet d'écrire dans un fichier des données présentes en mémoire physique.
- **déplacement** : permet de déplacer la position courante dans le fichier.

A.2.2 Gestion du stockage

A.2.2.1 Sous-système de stockage

L'organisation typique d'un disque est illustrée sur la figure A.2. Le disque est organisé en une pile de **plateaux** rotatifs découpés en **secteurs** qui constituent les unités élémentaires de lecture et d'écriture. Les accès sont effectués par l'une des multiples **têtes** supportées par un unique **bras**. Le parcours suivi par une tête lors de la rotation d'un plateau correspond à une **piste**. L'ensemble des pistes accessibles par les têtes pour un positionnement donné du bras est appelé **cylindre**.

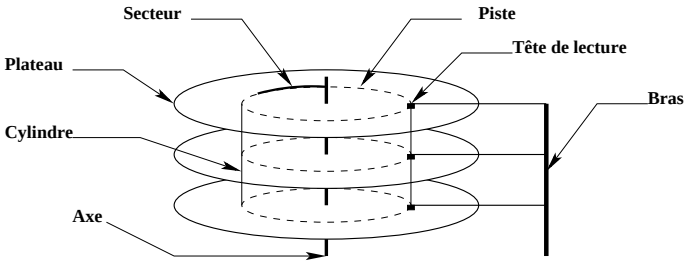

FIG. A.2 – *Organisation d'un disque dur*

Étant donné l'organisation du disque, un transfert disque doit spécifier le cylindre, la tête, le secteur de départ et le nombre de secteurs contiguës à lire ou écrire. Ces paramètres physiques sont masqués par les sous-systèmes de stockage modernes, grâce à la notion de

bloc. Le disque est assimilé à une séquence de **blocs disques** qui constituent les unités élémentaires d'accès.

A.2.2.2 Organisation d'un fichier sur disque

Lorsqu'un fichier est stocké sur disque il est découpé en blocs logiques, qui sont placés dans des blocs disques alloués dynamiquement. Ces blocs n'étant pas forcément consécutifs, il est nécessaire de mémoriser leur emplacement pour être en mesure de les retrouver ultérieurement. Le SGF utilise pour cela deux types d'information stockées sur le disque : les données effectives des fichiers et des **données de contrôle** permettant de gérer l'allocation et la localisation du contenu des fichiers. UNIX^{TM}stocke les données de contrôle dans des blocs particuliers appelés **i-nœud** et blocs d'index. Un i-nœud contient les informations relatives à un fichier (nom, date de création, propriétaire, etc.), des pointeurs vers les premiers blocs de données et trois pointeurs vers des blocs d'index à plusieurs niveaux, contenant l'emplacement des blocs de données restants du fichier (fig A.3).

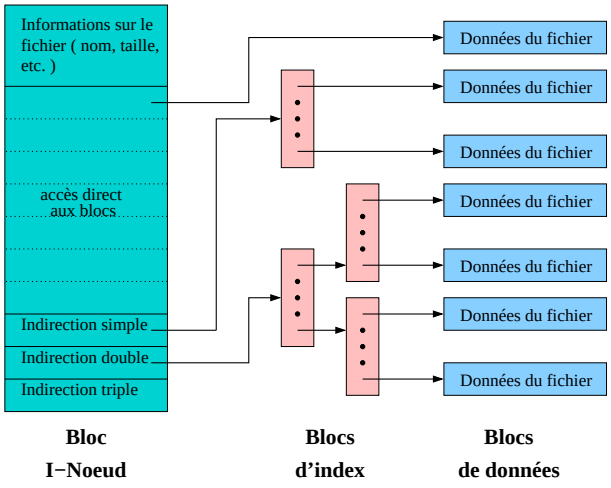

FIG. A.3 – *Structure d'un bloc i-nœud*

A.2.3 Accès aux fichiers

Lors d'un accès à une donnée d'un fichier, le SGF doit convertir cet accès logique **[1]** en lectures et écritures sur des blocs disques **[2]**. Cette conversion est réalisée en trois étapes (c.f. figure A.4).

Dans un premier temps, le SGF convertit le descripteur de fichier utilisateur **[3]** en numéro d'i-nœud **[4]** via la table des fichiers ouverts **[5]**. Ensuite, il effectue la correspondance entre la position de l'accès dans le fichier **[6]** et les blocs logiques correspondants **[7]**.

La deuxième étape consiste à déterminer la liste des blocs physiques **[8]** correspondant aux blocs logiques. Ceci est réalisé grâce à la **table d'implantation [9]**, qui contient la liste des correspondances entre blocs logiques et blocs physiques.

Enfin, le SGF demande la lecture ou l'écriture des blocs physiques au sous-système de stockage. Le résultat de l'opération est retourné à l'utilisateur.

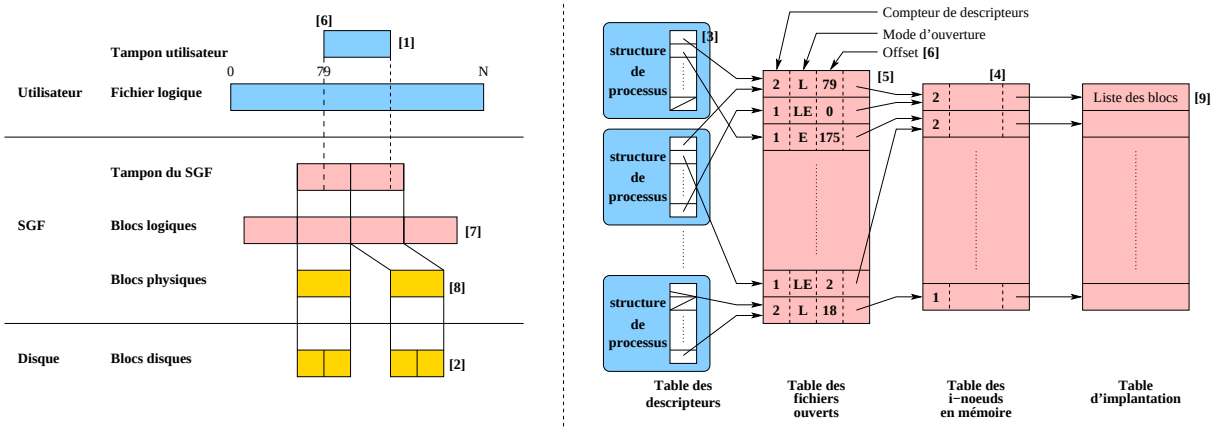

FIG. A.4 – *Gestion des accès disques*

A.2.4 Amélioration des performances

Les unités de stockage secondaires sont généralement des périphériques disposant de pièces mécaniques (bras de lecture, moteurs, etc) dont la réactivité est très faible comparée aux dispositifs électroniques d'un ordinateur. Ainsi, le temps de déplacement d'un bras de lecture ou de positionnement d'une tête au dessus d'un secteur particulier est de l'ordre de plusieurs millisecondes, à comparer aux quelques nanosecondes nécessaires à l'exécution d'une instruction sur un processeur. Le chargement de données depuis un disque représente donc un facteur de ralentissement très important pour une application et potentiellement pour tout le système.

Un disque souffre de deux problèmes majeurs de performance : une latence élevée (quelques millisecondes) et un faible débit (quelques méga-octets par seconde). Afin de masquer la latence et d'optimiser l'utilisation de la bande passante des disques, les SGF utilisent couramment plusieurs techniques d'optimisation.

A.2.4.1 Cache disque

Définition 39 (Cache disque) *Un **cache disque** est un dispositif logiciel assurant le maintien en mémoire physique des fichiers les plus fréquemment accédés.*

Le cache disque repose sur le principe de localité temporelle (voir chapitre 3.2.1). En gardant en mémoire les blocs les plus souvent référencés, le nombre d'accès réels aux disques est limité. Pour les blocs présents en cache, les accès sont réalisés en quelques microsecondes. De plus, les requêtes servies par le cache ne surchargent pas inutilement le disque, qui peut ainsi servir d'autres requêtes.

A.2.4.2 Pré-chargement

Définition 40 (Pré-chargement) *Le **pré-chargement** consiste à charger des données depuis une unité de stockage secondaire avant qu'elles ne soient effectivement demandées par l'utilisateur.*

Le pré-chargement repose sur le principe de localité spatiale (voir chapitre 3.2.1). En pratique, la plupart des fichiers sont lus séquentiellement [85]. Le SGF utilise cette propriété pour anticiper les requêtes et charger les blocs dans le cache avant que ceux-ci ne soient effectivement demandés par l'utilisateur. La latence d'accès aux blocs déjà chargés est ainsi fortement diminuée.

A.2.4.3 Écritures retardées

Avec un mécanisme d'écritures retardées, lorsqu'un utilisateur écrit dans un fichier, les données sont placées dans le cache et transmises réellement au disque après quelques secondes. Ceci permet d'effectuer des accès groupés et évite les écritures multiples sur un même bloc du disque. Cependant, les données présentes en cache et non encore sauvegardées sont perdues en cas d'arrêt brutal du système (coupure de courant, etc).

A.3 La gestion de processus

A.3.1 Notion de processus

Les premiers systèmes informatiques permettaient l'exécution d'un seul programme à la fois. Celui-ci possédait le contrôle total du système et l'accès à toutes les ressources de la machine. Les systèmes informatiques modernes permettent de charger en mémoire plusieurs programmes, afin de les exécuter en concurrence. Cette évolution a nécessité un contrôle plus ferme et plus de séparation des divers programmes. Ces besoins ont abouti à la notion de **processus**.

Définition 41 (Processus) *Un **processus** est un programme en cours d'exécution. Il représente une entité dynamique d'exécution qui peut être lancée sur le processeur et interrompue par le système d'exploitation.*

Un processus est défini par son contexte d'exécution : compteur ordinal, pile, variables globales, état du processus, etc. Toutes ces informations sont gérées par le système au moyen d'un **bloc de contrôle** ou « *Process Control Block (PCB)* ».

Sur une architecture mono-processeur, plusieurs processus peuvent coexister, mais un seul est en cours d'exécution à tout instant sur le processeur. Sur une architecture multi-processeurs plusieurs processus peuvent s'exécuter simultanément. Il est possible de placer un processus différent sur chaque processeur disponible à un instant donné. Cependant, un processus donné ne peut être exécuté sur plusieurs processeurs simultanément.

A.3.2 Ordonnancement de processus

Grâce à la notion de processus, il est possible de partager le temps processeur entre différents processus, c'est le **temps partagé** ou « ***time sharing*** ». Ceci permet de ne pas laisser le processeur inactif lors de l'accès bloquant à un périphérique et donc d'utiliser cette ressource au maximum.

Une tâche particulière, l'ordonnanceur, est spécialisée dans le choix du processus à exécuter. Il dispose d'une **file d'attente** de processus, contenant la liste des programmes chargés en mémoire et devant être exécutés. L'ordonnanceur est activé périodiquement : il choisit un processus prêt, lit dans le PCB les informations nécessaires à la reprise et active le processus. Ce processus est interrompu après un temps décidé par l'ordonnanceur, ou s'il est en attente d'entrées/sorties. Lors de l'arrêt d'un processus, son état est sauvegardé et l'ordonnanceur choisit un autre processus à exécuter. C'est le **changement de contexte**.

B ALGORITHMES DE GESTION DE LA COHÉRENCE

B.1 Notations

Pour désigner les principales structures de données, nous utiliserons les notations suivantes :

propriétaire[c,p] : identité du nœud propriétaire de la page p du conteneur c. Table gérée par les gestionnaires de pages.

état[c,p] : droit d'accès à la page p du conteneur c sur le nœud local. Table gérée par chaque nœud.

copie[c,p] : liste des nœuds disposant d'une copie de la page p du conteneur c. Table gérée par les propriétaires de pages.

NodeId : identifiant du nœud local.

LieurES[c] : lieur d'entrée/sortie associé au conteneur c. Table dupliquée sur chaque nœud.

LieurInterface[c] : lieur d'interface associé au conteneur c. Table dupliquée sur chaque nœud.

Attachement[c] : nœuds attaché au conteneur c. Table dupliquée sur chaque nœud.

B.2 La fonction Get_Page sans les mécanismes de lieurs

fonction **GET_PAGE (*c* : identifiant du conteneur ; *p* : identifiant de la page)**
{
    Envoyer (copie_lecture,c,p,NodeId) au gestionnaire de la page *(c,p)* ;
    Allouer un cadre de page ;
    Recevoir copie de la page *(c,p)* dans le cadre de page alloué ;
    état[c,p] := lecture ;
    Retourner adresse de la page *(c,p)* ;
}

Sur réception du message (copie_lecture,c,p,N) sur le nœud gestionnaire :
{
    Envoyer message (copie_lecture,c,p,N) au nœud *propriétaire[c,p]* ;
}

Sur réception du message (copie_lecture,c,p,N) sur le nœud propriétaire :
{
    copie[c,p] := *copie[c,p]* + {N} ;
    Envoyer copie de la page *(c,p)* au nœud N ;
    état[c,p] := lecture ;
}

B.3 La fonction Grab_Page sans les mécanismes de lieurs

***fonction* GRAB_PAGE (c : identifiant du conteneur ; p : identifiant de la page)**
{
Envoyer message (copie_écriture,c,p,NodeId) au nœud gestionnaire de la page (*c,p)* ;
Allouer un cadre de page ;
Recevoir copie de la page (*c,p)* dans le cadre de page alloué ;
état[c,p] := lecture-écriture ;
copie[c,p] := {NodeId} ;
Envoyer (acquittement) au nœud gestionnaire de la page (*c,p)* ;
Retourner adresse de la page (*c,p)* en mémoire locale ;
}

Sur réception du message (copie_écriture,c,p,N) sur le nœud gestionnaire :
{
Envoyer message (copie_écriture,c,p,N) au nœud *propriétaire[c,p]* ;
propriétaire[c,p] := N ;
Recevoir (acquittement) du nœud N ;
}

Sur réception du message (copie_écriture,c,p,N) sur le nœud propriétaire :
{
pour tous les nœuds i dans *copie[c,p]* **faire**
Envoyer message (invalidation,c,p,NodeId) au nœud i;
fin pour
pour tous les nœuds i dans *copie[c,p]* **faire**
Recevoir (acquittement) du nœud i;
fin pour
copie[c,p] := {} ;
Envoyer copie de la page (*c,p)* au nœud N ;
}

Sur réception du message (invalide,c,p,N) sur un nœud :
{ Obtention d'une copie en écriture GRABPAGE4
état[c,p] := invalide ;
Libérer cadre de page de la page (*c,p)* ;
Envoyer message (acquittement) au nœud N ;
}

B.4 La fonction Get_Page avec les mécanismes de lieurs

fonction **GET_PAGE** (*c* **: identifiant du conteneur ;** *p* **: identifiant de la page)**
{
Envoyer (copie_lecture,c,p,NodeId) au gestionnaire de la page *(c,p)* ;
Allouer un cadre de page ;
Recevoir copie de la page *(c,p)* dans le cadre de page alloué ;
si message reçu = (premier_accès_local) **alors**
Libérer le cadre de page ;
LieurES[c].first_ touch(c,p) ;
fin si
état[c,p] := lecture ;
LieurInterface[c].change_ access(c,p,lecture) ;
Retourner adresse de la page *(c,p)* ;
}

Sur réception du message (copie_lecture,c,p,N) sur le nœud gestionnaire :
{
si *propriétaire[c,p]* = aucun **alors**
si *Attachement[c]* = Aucun **ou** *Attachement[c]* = N **alors**
Envoyer message (premier_accès_local) au nœud N ;
propriétaire[c,p] := N ;
sinon
Envoyer message (premier_accès_lecture,c,p,N) au nœud *Attachement[c]* ;
propriétaire[c,p] := *Attachement[c]* ;
fin si
sinon
Envoyer message (copie_lecture,c,p,N) au nœud *propriétaire[c,p]* ;
fin si
}

Sur réception du message (premier_accès_lecture,c,p,N) sur le nœud attaché :
{
copie[c,p] := {NodeId,N} ;
LieurES[c].first_ touch(c,p) ;
Envoyer copie de la page *(c,p)* au nœud N ;
}

B.5 La fonction Grab_Page avec les mécanismes de lieurs

***fonction* GRAB_PAGE (c : identifiant du conteneur ; p : identifiant de la page)**
{
Envoyer message (copie_lecture,c,p,NodeId) au nœud gestionnaire de la page *(c,p)* ;
Allouer cadre de page ;
Recevoir copie de la page *(c,p)* dans le cadre de page alloué ;
si message reçu = (premier_accès_local) **alors**
Libérer le cadre de page ;
LieurES[c].first_touch(c,p) ;
fin si
état[c,p] := lecture-écriture ;
LieurInterface[c].change_access(c,p,lecture-écriture) ;
copie[c,p] := {NodeId} ;
Envoyer (acquittement) au nœud gestionnaire de la page *(c,p)* ;
Retourner adresse de la page *(c,p)* ;
}

Sur réception du message (copie_écriture,c,p,N) sur le nœud gestionnaire :
{
si *propriétaire[c,p]* = aucun **alors**
si *Attachement[c]* = Aucun **ou** *Attachement[c]* = N **alors**
Envoyer message (premier_accès_local) au nœud N ;
sinon
Envoyer message (premier_accès_écriture,c,p,N) au nœud *Attachement[c]* ;
fin si
sinon
Envoyer message (copie_écriture,c,p,N) au nœud *propriétaire[c,p]* ;
fin si
propriétaire[c,p] := N ;
Recevoir (acquittement) du nœud N ;
}

Sur réception du message (premier_accès_écriture,c,p,N) sur le nœud attaché :
{
LieurES[c].first_touch(c,p) ;
Envoyer copie de la page *(c,p)* au nœud N ;
LieurES[c].invalidate_page(c,p) ;
}

Sur réception du message (invalide,c,p,N) sur un nœud :
{
état[c,p] := invalide ;

LieurES[c].invalidate_page(c,p) ;
LieurInterface[c].change_access(c,p,invalide) ;
Envoyer message (acquittement) au nœud N ;
}

C ALGORITHMES DE GESTION DU REMPLACEMENT DE PAGE

C.1 Eviction d'un duplicata

***fonction* ÉVINCER_DUPLICATA (c : identifiant du conteneur ; p : identifiant de la page)**
{
    Libérer le cadre de la page *(c,p)* ;
    état[c,p] := invalide ;
    Envoyer (éviction_copie,c,p,NodeId) au nœud gestionnaire de la page *(c,p)* ;
}

Sur réception du message (éviction_copie,c,p,N) sur le nœud gestionnaire :
{
    Envoyer message (éviction_copie,c,p,N) au nœud *Attachement[c]* ;
}

Sur réception du message (éviction_copie,c,p,N) sur le nœud propriétaire :
{
    copie[c,p] := *copie[c,p]* - {N} ;
}

C.2 Migration de la propriété d'une page

```
fonction MIGRER_PROPRIÉTÉ (c : identifiant du conteneur ; p : identifiant de la page)
{
    copie[c,p] := copie[c,p] - {NodeId} ;
    tant que copie[c,p] != 0 faire
        Choisir N dans copie[c,p] ;
        Envoyer (migre_propriété,c,p,NodeId, copie[c,p]) au nœud N ;
        Attendre acquittement ;
        si Message reçu = (acquittement) alors
            copie[c,p] := 0 ;
            état[c,p] := invalide ;
            Libérer le cadre de la page (c,p) ;
        sinon
            copie[c,p] := copie[c,p] - {N} ;
        fin si
    fin tant que
    si état[c,p] != invalide alors
        Injecter_page(c,p) ;
    fin si
}

Sur réception du message (migre_propriété,c,p,N,copie) sur un nœud :
{
    si état[c,p] = invalide alors
        Envoyer (acquittement_négatif) au nœud N ;
    sinon
        copie[c,p] := copie ;
        Envoyer (migre_propriété,c,p,N,NodeId) au nœud gestionnaire de la page (c,p) ;
    fin si
}

Sur réception du message (migre_propriété,c,p,Old,New) sur le nœud gestionnaire :
{
    propriétaire[c,p] := New ;
    Envoyer (acquittement) au nœud Old ;
}
```

C.3 Injection d'une page

***fonction* INJECTER_PAGE (c : identifiant du conteneur ; p : identifiant de la page)**
{
N = choisir_noeud() ;
Envoyer (Injection_page,c,p,page (*c,p)*) au nœud N ;
Attendre acquittement ;
copie[c,p] := ;
état[c,p] := invalide ;
Libérer le cadre de la page (*c,p)* ;
}

Sur réception du message (Injection_page,c,p,N,page) sur un nœud :
{
Allouer un cadre de page ;
Placer la page reçue dans le cadre de page ;
copie[c,p] := NodeId ;
état[c,p] := lecture-écriture ;
Envoyer (injection_page,c,p,N,NodeId) au nœud gestionnaire de la page (*c,p)* ;
}

Sur réception du message (injection_page,c,p,Old, New) sur le nœud gestionnaire :
{
propriétaire[c,p] := New ;
Envoyer (acquittement) au nœud Old ;
}

C.4 Remplacement de page sur lieur d'entrée/sortie

***fonction* REMPLACER_PAGE (c : identifiant du conteneur ; p : identifiant de la page)**
{
    si *Attachement[c]* = Aucun **alors**
        LieurES[c].flush_page(c,p) ;
    sinon
        Envoyer message (Remplacer_page,c,p,NodeId,page) au nœud *Attachement[c]* ;
        Attendre (acquittement) ;
    fin si
    copie[c,p] := ;
    état[c,p] := Invalide ;
    Libérer le cadre de la page *(c,p)* ;
}

Sur réception du message (Remplacer_page,c,p,N,page) sur un nœud :
{
    LieurES[c].flush_page(c,p) ;
    Envoyer message (Acquitte_Remplacer,c,p,N) au nœud gestionnaire de la page *(c,p)* ;
}

Sur réception du message (Acquitte_Remplacer,c,p,N) sur le nœud gestionnaire :
{
    propriétaire[c,p] := Aucun ;
    Envoyer (acquittement) au nœud N ;
}

D LE SYSTÈME D'EXPLOITATION LINUX

D.1 Modules

Le concept de module offert par Linux permet d'ajouter de nouvelles fonctionnalités au noyau sans avoir à modifier son code source. Il est ainsi inutile de recompiler le noyau et de redémarrer la machine pour pouvoir utiliser ces nouvelles fonctionnalités. Un module peut être chargé et déchargé à chaud, c'est-à-dire pendant que le système est en fonctionnement. Pour charger et décharger un module, les commandes systèmes *insmod* et *rmmod* sont utilisées. Celles-ci nécessitent les privilèges du super-utilisateur.

Lorsque le module est chargé, le code assembleur correspondant est ajouté à celui du noyau. Les fonctions du noyau utilisées par le module sont liées grâce à la table des symboles du noyau. Une fois le module placé en mémoire et lié au code du noyau, une fonction d'initialisation du module est exécutée. Cette fonction doit obligatoirement être présente dans le module. Après l'exécution de cette fonction, le module fait partie intégrante du noyau.

La notion de module a plusieurs avantages. Le premier est de pouvoir ajouter de nouvelles fonctionnalités au noyau sans nécessiter le redémarrage de la machine. Cette fonctionnalité est très utilisée pour charger des pilotes de périphériques, ce qui permet de changer les versions de pilotes sans modifier le noyau. Un deuxième avantage est de pouvoir réaliser des développements au sein du noyau plus rapidement. En supprimant l'étape de redémarrage du système pour tester les modifications, un gain de temps significatif est obtenu. Enfin, le code du noyau LINUX évolue beaucoup d'une version à l'autre. Ajouter des fonctionnalités au sein même du code du noyau implique généralement de les réécrire pour chaque nouvelle version. Les modules s'interfacent au noyau grâce à un ensemble de primitives qui assurent une certaine indépendance vis-à-vis des modifications des sources du noyau. Les fonctionnalités ajoutées au sein de modules ont ainsi une durée de vie plus élevée et peuvent plus facilement être mises à jour.

D.2 Gestion mémoire

Le système de gestion mémoire de LINUX gère la mémoire grâce à deux types d'abstraction : (1) l'espace d'adressage des processus et (2) les régions de mémoire virtuelle (voir figure D.1). A chaque processus est associé un espace d'adressage de 32 bits couvrant quatre giga-octets **[1]**. Les trois premiers giga-octets contiennent les données du processus tandis que le dernier giga-octet contient le texte et les données du noyau. Chaque processus

dispose de son propre espace d'adressage représenté au sein du noyau grâce à une table des pages à trois niveaux [1]. Sur les architectures x86, la taille des pages est de 4 Ko.

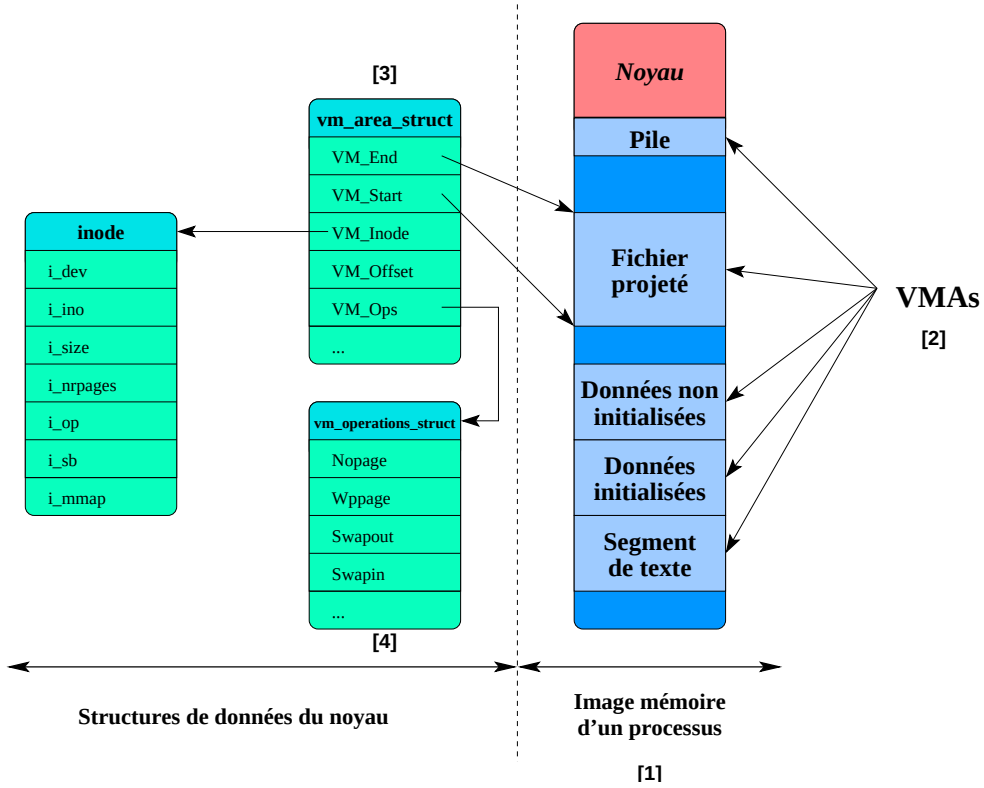

FIG. D.1 – *Gestion de l'image mémoire d'un processus dans Linux*

Au sein de l'espace d'adressage d'un processus, le noyau distingue des **région de mémoire virtuelle** ou « *Virtual Memory Area (VMA)* » **[2]**. Chaque VMA est caractérisée par son adresse de début, sa taille, ses droits d'accès et le fichier associé. Une VMA correspond à la mise en œuvre du concept de segment de mémoire. Il existe une VMA différente pour chaque segment de texte, de donnée, de pile, etc. Le segment de texte est mis en œuvre par une VMA associée au fichier contenant le texte de l'application. Le segment de données est mis en œuvre par une VMA liée uniquement à une zone de mémoire physique. Une VMA peut être partagée par différents processus, afin d'économiser des cadres de pages ou de partager des données entre *threads*.

La structure de donnée *vm_area_struct* **[3]** contient les informations de gestion d'une VMA. Le champ *vm_ops* pointe vers une structure de données *vm_operations_struct* **[4]** contenant une liste de pointeurs sur fonctions. Ces fonctions représentent les opérations élémentaires de gestion d'une VMA : allocation d'une page en cas de premier accès (*nopage*), copie d'une page sur écriture (*wppage*), remplacement d'une page (*swapout*), etc. A

[1] Les processeurs de la famille x86 ne disposant que de 2 niveaux de table des pages, le niveau intermédiaire au sein des structures du noyau n'est pas utilisé.

chaque VMA est associée une liste d'opérations qui peuvent être modifiées par le noyau ou à travers un module, afin de changer le comportement de la VMA.

Enfin, le champs *vm_inode* pointe vers la structure *inode*, contenant les informations relatives au fichier associé à la VMA. Une VMA peut cependant n'être liée à aucun fichier, c'est notamment le cas des VMAs contenant les segments de données tel que le tas, la pile, etc.

D.3 Système de fichiers

Afin de permettre la cohabitation de plusieurs systèmes de fichiers au sein du noyau et faciliter la création et l'utilisation de nouveaux systèmes de fichiers, LINUX utilise le concept de système de fichiers virtuel ou « *Virtual File System (VFS)* » (voir figure D.2, **[1]**). Le VFS se présente sous la forme d'une couche logicielle d'abstraction des systèmes de fichiers sous-jacents, en offrant une interface générique de type lecture/écriture **[2]** à l'ensemble de ces systèmes de fichiers **[3]**. Les systèmes de fichiers les plus couramment rencontrés sous LINUX sont « *Second Extended File System (Ext2)* », qui est le système de fichier natif de LINUX, « *FAT32* » ou « *NTFS* » les systèmes de fichiers de Microsoft, et plus récemment « *Ext3* » ou « *Raiser FS* », des systèmes de fichiers journalisés mis en œuvre pour le système LINUX.

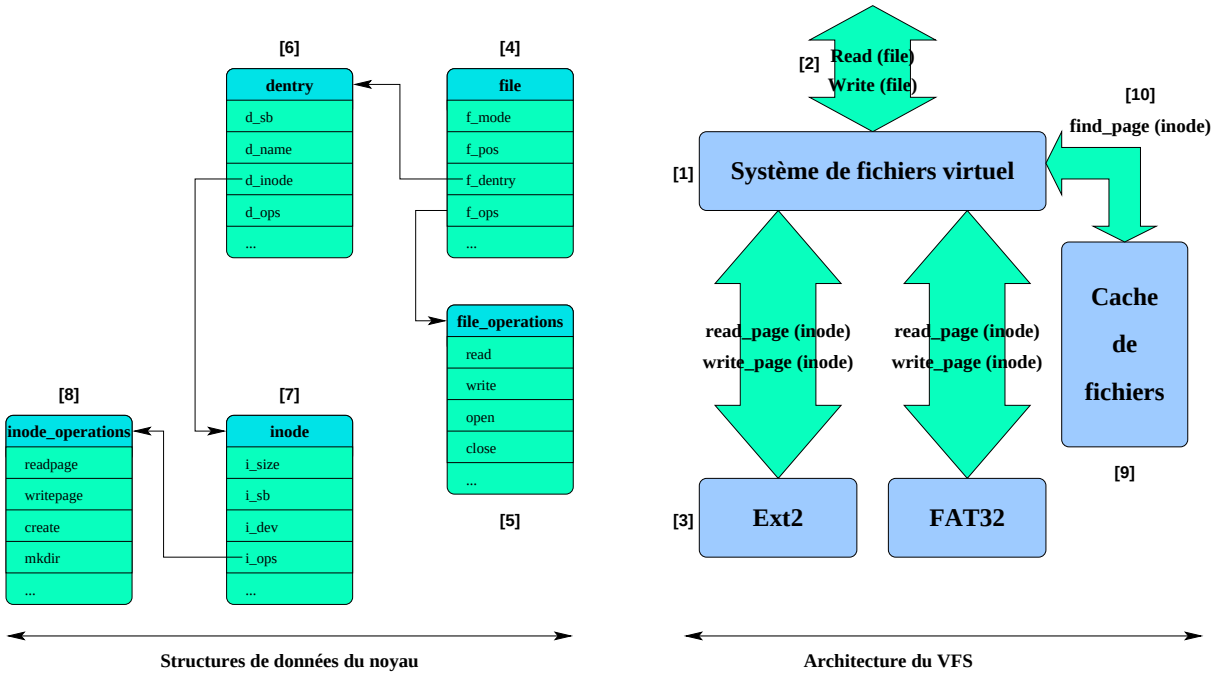

FIG. D.2 – *Gestion des fichiers dans Linux*

Au sein du VFS, un fichier ouvert est représenté par une structure *file* **[4]**, contenant entre autre le mode d'ouverture, la position courante dans le fichier, un pointeur vers une

structure *dentry* et un pointeur vers une table de pointeurs sur fonctions **[5]**. Cette table contient des pointeurs sur les fonctions de base de manipulation des fichiers : ouverture, lecture, écriture, etc. Ces fonction peuvent être modifiées par chaque système de fichiers.

La structure *dentry* contient entre autre, un pointeur sur le super-bloc du système de fichier associé au fichier, le nom du fichier, et un pointeur sur l'i-nœud du fichier **[7]** et un pointeur vers une table de pointeurs sur fonctions. Une structure *dentry* est utilisée pour chaque lien vers un même fichier physique.

La structure *inode* contient entre autre un pointeur vers une table de pointeurs sur fonctions **[8]**. Cette table contient des pointeurs sur les fonctions de base de manipulation des i-nœuds : lecture ou écriture d'une page de données, création d'un nouvel i-nœud, etc. Ces fonction sont spécifiques à chaque système de fichiers. Une structure *inode* est utilisée pour chaque fichier en cours d'utilisation.

Lors d'un appel à une fonction d'accès à un fichier, la fonction *read* par exemple, le VFS détermine quel est le système de fichiers qui est en jeu et exécute la fonction le lecture associée à ce système de fichier, située dans la table *file_ operations*. Cette fonction convertit l'accès à une suite d'octets en accès à un ensemble de pages et appelle la fonction *read_page* située dans la table *inode_ operations* associée à l'i-nœud du fichier afin de charger ces pages.

Afin de limiter le nombre d'accès aux disques, le VFS utilise un cache de fichiers **[9]**. Avant chaque appel à la fonction *read_page*, un appel à la fonction *find_page* permet de vérifier si la page à charger est présente dans le cache. Si la page est présente dans le cache, aucun accès disque n'est réalisé.

D.4 Processus et threads

Au sein du noyau LINUX, l'unité d'exécution est le **processus**. Un processus est représenté dans le noyau par une structure nommée *task_ struct* (voir figure D.3 **[1]**). Celle-ci contient de nombreux champs, dont le nom du processus, son identifiant, son état (actif, stoppé, terminé, etc), des pointeurs vers ses pères et fils, un pointeur vers une structure *mm_ struct* **[2]** décrivant l'espace d'adressage du processus et notamment la liste de ses VMAs **[3]** et un pointeur sur une structure *files_ struct* **[4]** regroupant la liste des fichiers ouverts par le processus **[5]**.

La notion de processus léger n'existe pas dans le noyau du système LINUX. Cependant, certaines particularités du noyau permettent de mettre en œuvre des activités comparables aux processus légers. Ces activités appelées *threads* par abus de langage correspondent en réalité à des processus lourds partageant une partie de leur espace d'adressage grâce au partage de VMAs. Ces processus ou *threads* peuvent ainsi partager naturellement des variables situées en mémoire, mais ne sont pas « *légers* » au sens ou des structures de données complexes leur sont associées dans le noyau et que le changement de *thread* actif implique un changement de contexte de processus, ce qui est une opération lourde et coûteuse.

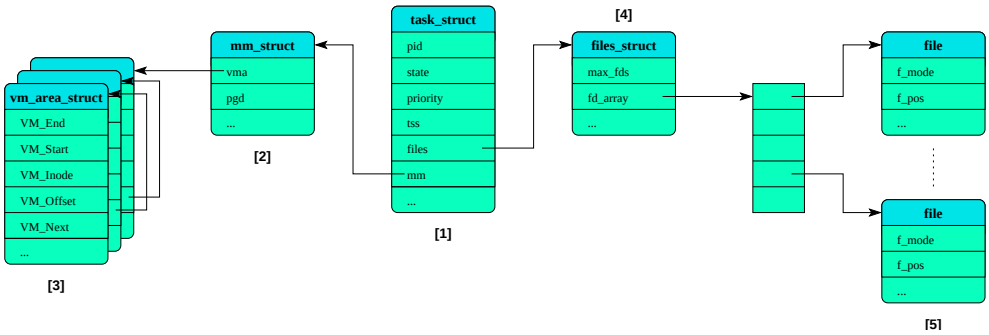

FIG. D.3 – *Gestion des processus dans Linux*

E MISE EN ŒUVRE DU MODULE GIMLI

E.1 Mise en œuvre de Gimli

La mise en œuvre du système de communication GIMLI doit répondre à deux critères apparemment contradictoires : performance et indépendance vis-à-vis de la technologie de communication. GIMLI doit en effet être capable d'envoyer et de recevoir des messages rapidement, c'est-à-dire avec une latence faible et un débit élevé, mais doit également pouvoir fonctionner sur différentes technologies réseaux en conservant la même interface de haut niveau.

Pour assurer le critère de performance, une implémentation efficace et de bas niveau est nécessaire. Efficace, en optimisant les opérations effectuées sur les messages et en réduisant le nombre de copies des données à transmettre. De bas niveau, en se plaçant très près du matériel afin de limiter le surcoût de traversé des couches logicielles.

Pour que GIMLI puisse fonctionner sur différentes technologies réseaux, il est nécessaire de réécrire une partie du code pour chaque type de matériel. Afin de minimiser cette réécriture, un découpage en couches a été utilisé pour isoler la partie fortement dépendante du matériel. Enfin, afin de limiter la complexité et le temps de développement, les couches dépendantes du matériel sont issues de la bibliothèque de communication GAMMA [23] sur carte Ethernet et de la bibliothèque de communication Bip [82] sur carte Myrinet.

E.1.1 Interface de communication

La bibliothèque de communication GIMLI offre deux types de communications : l'échange de messages à travers une interface de type envoi/réception et les messages actifs. Chaque message transmis via GIMLI est identifié grâce à un **profil** composé de 3 paramètres :

- **nœud destinataire / émetteur** : il s'agit du numéro logique du nœud destinataire/émetteur du message. Ce numéro logique est interne à GIMLI et compris entre 0 et N-1, N étant le nombre de nœuds de la grappe. En réception, il est possible d'attendre un message en provenance de n'importe quel nœud grâce à la valeur *-1*.
- **type** : le type de message est une valeur associée au message qui permet de lui donner une sémantique particulière. La valeur de ce paramètre est laissée au choix de l'utilisateur. En réception, il est possible d'attendre un message de n'importe quel type grâce à la valeur *-1*.
- **port** : le port est un niveau supplémentaire de typage. Il permet par exemple de diriger un ensemble de messages vers une entité logicielle particulière. Le port doit

toujours être fixé, en émission comme en réception.
L'interface de GIMLI est composée des fonctions suivantes :

GOBELINS_SEND(nœud, type, port, buffer, taille) ;
GOBELINS_RECV(nœud, type, port, buffer, taille, status) ;
GOBELINS_SEND_AM(nœud, type, buffer, taille) ;
GOBELINS_BROADCAST(identifiant de groupe, type, port, buffer, taille) ;
GOBELINS_FAST_BROADCAST(liste de nœuds, type, port, buffer, taille) ;

E.1.2 Architecture de Gimli

La figure E.1 présente une vue générale de l'architecture du système GIMLI, découpée en quatre couches : une couche offrant l'API utilisateur, une couche permettant de stocker les messages en attente de traitement, une couche assurant le contrôle d'erreur et une couche bas niveau assurant l'accès au réseau.

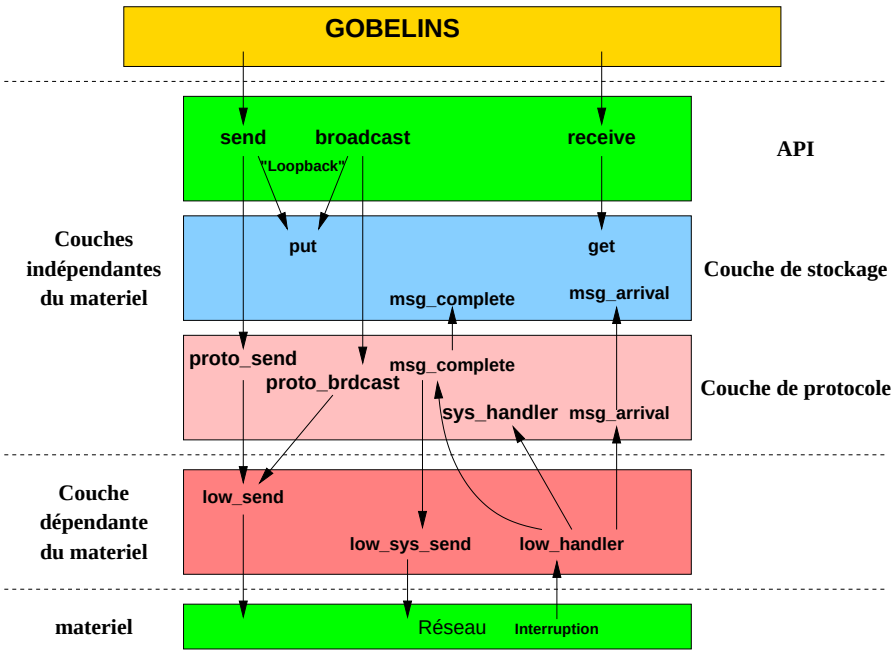

FIG. E.1 – *Vue générale du système* GIMLI

E.1.2.1 L'API

La couche d'API met en œuvre l'interface utilisateur et fait le lien avec la couche de stockage et la couche d'accès au réseau. Elle offre les fonctions d'interface décrites précédemment. Les fonctions d'émission transmettent directement les messages à la couche de protocole si le destinataire du message n'est pas le nœud local. Dans le cas d'un envoie

de message au nœud local ou dans le cas d'une réception, la requête est transmise à la couche de stockage.

E.1.2.2 Couche de stockage

Lors de l'émission/réception asynchrone de messages il est fréquent qu'un message arrive sur un nœud avant qu'une demande de réception ne soit posée ou à l'inverse, qu'une demande de réception ne soit posée avant que le message correspondant ne soit arrivé.

Dans le cas où une requête de réception est posée sur un message non encore arrivé, la requête est stockée en attendant l'arrivée du message. Dans le cas inverse, où un message arrive avant que l'utilisateur n'ait explicitement demandé sa réception, celui-ci est stocké jusqu'à demande de réception par l'utilisateur. L'interface d'accès à la couche de stockage est composée de quatre fonctions :

- ***put_ message*** : cette fonction permet de stocker un message émis en « *loopback* »[1]. Dans le cas où le message est déjà attendu par un processus, le message n'est pas stocké mais directement envoyé à ce processus.
- ***get_ message*** : cette fonction permet de stocker une requête de réception de message. Si le message attendu a déjà été reçu et placé dans la couche de stockage, la requête n'est pas stockée et le message correspondant est transmis au processus demandeur.
- ***notify_ msg_ arrival*** : cette fonction permet de notifier à la couche de stockage l'arrivée d'un nouveau message. Si un processus est en attente de ce message, le tampon de réception associé est retourné à la couche de protocole. Sinon, un tampon de réception est alloué et son adresse est retournée à la couche de protocole.
- ***notify_ msg_ complete*** : cette fonction permet de notifier que le message dont l'arrivée a été signalé grâce à la fonction *notify_ msg_ arrival* a été placé dans le tampon de réception. Si un processus en attente de ce message se trouve dans la couche de stockage, le message lui est transmis.

E.1.2.3 Couche de protocole

La couche de protocole se charge d'empaqueter le profil du message et d'autres informations utiles au fonctionnement de la bibliothèque (numéro de séquence, CRC, etc) dans le tampon à envoyer. La couche de protocole assure également la retransmission de messages en cas de perte ou d'erreur d'intégrité. L'interface d'accès à la couche de stockage est composée de six fonctions :

- ***gimli_ proto_ send*** : cette fonction permet de transmettre un message. Elle place en tête du message son profil, son numéro de séquence et des données de contrôle d'erreur et transmet le tampon à la fonction *gimli_ low_ send* de la couche basse de communication. Une fois le message transmis à la couche de bas niveau, la fonction attend un acquittement du nœud destinataire. En cas d'acquittement négatif ou de « *time out* », le message est retransmis.

[1] nœud émetteur = nœud destinataire

- ***gimli_proto_broadcast*** : cette fonction permet de diffuser un message vers un groupe de nœuds et fonctionne de manière similaire à la fonction *gimli_proto_send*. La différence réside dans la construction de l'entête. De plus, il n'y a pas de contrôle d'erreur dans la version actuelle.
- ***gimli_proto_fast_broadcast*** : cette fonction permet de diffuser un message vers un ensemble de nœuds. Elle fonctionne de la même manière que la fonction *gimli_proto_send*. La différence réside dans la construction de l'entête. De plus, il n'y a pas de contrôle d'erreur dans la version actuelle.
- ***gimli_proto_notify_msg_arrival*** : cette fonction est appelée par la couche de bas niveau lorsqu'un message arrive sur la carte. Cette fonction vérifie que le numéro de séquence du message est correct. Si c'est le cas, elle appelle la fonction *notify_msg_arrival* pour obtenir un tampon de réception qui est retournée la couche bas niveau. S'il y a un problème de séquence, un code d'erreur est retourné pour signifier à la couche bas niveau que le message doit être ignoré.
- ***gimli_proto_notify_msg_complete*** : cette fonction est appelée par la couche de bas niveau lorsqu'un message a entièrement été placé dans le tampon de réception. La fonction vérifie l'intégrité du message. Si le message est correct, il est envoyé à la fonction *notify_msg_complete* et un acquittement est envoyé au nœud émetteur. Si le message est corrompu, un acquittement négatif est envoyé au nœud émetteur pour demander une retransmission.
- ***gimli_proto_sys_handler*** : cette fonction traite les messages express utilisés pour transmettre les acquittements utilisés dans le protocole de contrôle d'erreurs. Elle est appelée par la couche basse lors de l'arrivée d'un message express.

E.1.2.4 Couche de bas niveau

La couche de bas niveau réalise l'accès physique au réseau. Cette partie est mise en œuvre en partie dans le pilote de la carte réseau, elle est donc fortement dépendante du matériel. Les boucles principales d'émission et de réception sont dépendantes du type de technologie utilisée (Ethernet, Myrinet, etc) alors que les fonctions d'accès physique à la carte réseau sont dépendantes du modèle de la carte réseau. L'interface de cette couche est composée de deux fonctions :

- ***gimli_low_send*** : cette fonction permet d'envoyer un message sur le réseau. Elle se charge de concatener l'entête GIMLI au message et de le découper en trames si nécessaire. Si le destinataire du message est *BROADCAST*, la couche de bas niveau doit se charger de diffuser le message sur le réseau.
- ***gimli_low_sys_send*** : cette fonction permet d'envoyer un message express. Cette fonction est utilisée uniquement au sein de GIMLI afin de transmettre des messages de contrôle comme l'envoi d'un acquittement de réception ou une demande de retransmission.

Lors de l'arrivée d'un message sur la carte, le « *handler* » de la couche basse fait appel à la fonction *gimli_proto_sys_handler* de la couche de protocole s'il s'agit d'un message express. S'il s'agit d'un message ordinaire, la couche de protocole fait appel à la fonction

gimli_proto_notify_msg_arrival pour obtenir un tampon dans lequel stocker le message. Si la fonction retourne une valeur négative, le message doit être ignoré.

Lorsqu'un message a été entièrement reçu et stocké dans le tampon de réception, le « *handler* » de la couche base fait appel à la fonction *gimli_proto_notify_msg_complete* de la couche de protocole pour qu'il soit traité et expédié aux couches supérieures.

E.1.3 Principe de fonctionnement

E.1.3.1 Émission d'un message

La figure E.2 présente les mécanismes mis en jeu durant l'émission d'un message. Dans le cas d'un message en « *loop-back* », la fonction *put_message* est appelée directement sans passer par la couche réseau **[1]**.

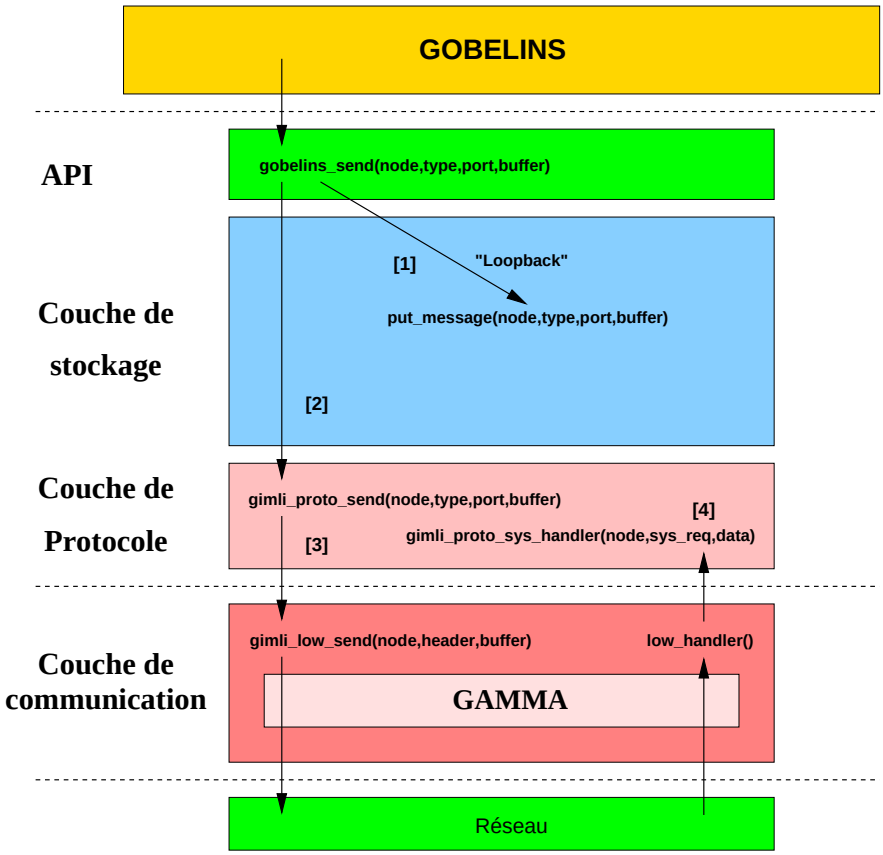

FIG. E.2 – *Émission d'un message*

Dans le cas d'un envoi distant, la requête d'émission est passée à la fonction *gimli_proto_send* de la couche de protocole **[2]**. Après empaquetage du message, celui-ci est transmis à la fonction *gimli_low_send* de la couche basse **[3]** pour être expédié sur le réseau et un acquittement est attendu en attente active **[4]**. Si un acquittement positif est reçu,

l'émission se termine. Si un acquittement négatif est reçu ou si l'attente de l'acquittement se termine sur un « *time-out* », le message est retransmis. Au bout de quatre tentatives infructueuses, un message d'erreur (-ETIME) est retourné.

L'acquittement est reçu et traité par la fonction *gimli_proto_sys_handler* qui débloque la fonction *gimli_proto_send* bloquée en attente active sur la variable *ack_spinlock*.

E.1.3.2 Réception d'un message

La figure E.3 présente les mécanismes mis en jeu durant la réception d'un message. Une réception est décomposée en deux parties. Tout d'abord, la demande de réception d'un message effectuée à la demande de l'utilisateur (fig. E.3.a). Dans le cas où le message a déjà été reçu, la fonction *get_message* retourne immédiatement le tampon reçu. Sinon, le processus ayant effectué la requête est endormi jusqu'à réception du message.

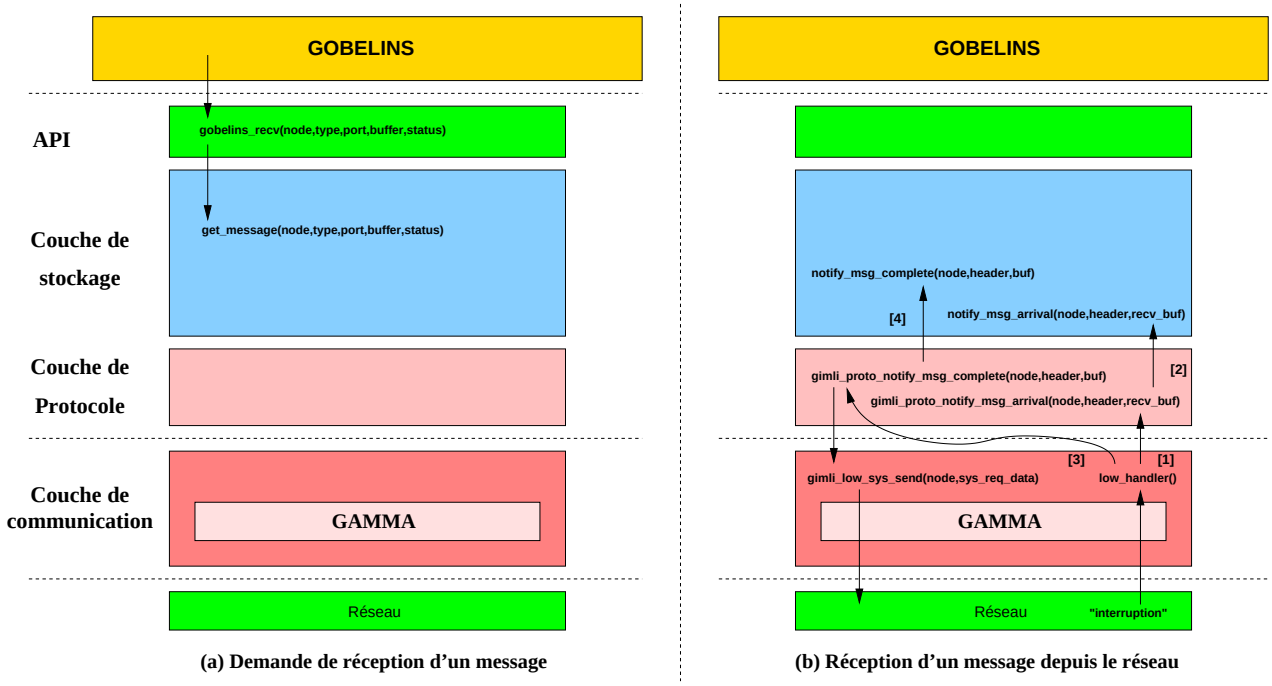

FIG. E.3 – *Réception d'un message*

Lorsqu'un message arrive de la carte réseau, une interruption déclenche le « *handler* » de la couche base de communication qui transmet le message à la fonction *gimli_proto_notify_msg_arrival* de la couche de protocole (voir figure E.3.b **[1]**). Celle-ci vérifie que le numéro de séquence du message est correct. Si c'est le cas, elle appelle la fonction *notify_msg_arrival* pour obtenir un tampon de réception **[2]**. Si le séquencement est incorrect, un code d'erreur est retourné à la couche basse.

Lorsque la fonction *notify_msg_arrival* reçoit la notification de l'arrivée d'un message, elle extrait le profil du message et vérifie si un processus est en attente de celui-ci. Si c'est

le cas, le tampon de réception associé est retourné. Sinon, un tampon est alloué et son adresse est retournée à la fonction de la couche de protocole.

Lorsqu'un message a été entièrement reçu, le « *handler* » de la couche basse appelle la fonction *gimli_proto_notify_msg_complete* **[3]** qui extrait le profil du message et vérifie l'intégrité du message. Si le message est correct, un acquittement positif est transmis à l'expéditeur et le message est passé à la fonction *notify_msg_complete* de la couche de stockage **[4]**. Si le message est incorrect, un acquittement négatif est envoyé à l'expéditeur et le message est détruit.

Lorsque la fonction *notify_msg_complete* de la couche de stockage reçoit une notification de réception, elle vérifie si un processus est en attente de ce message et le réveille. Si aucun processus n'est en attente, aucune opération n'est effectuée.

E.2 Étude de performance de la bibliothèque de communication Gimli

Nous avons mené plusieurs séries de tests afin de mesurer la latence et le débit obtenus avec la bibliothèque de communication GIMLI sur les réseaux Ethernet 100 Mb et Gigabit Ethernet. A ce jour, la mise en œuvre sur réseau Myrinet n'est pas terminée. Ses performances n'ont donc pas pu être évaluées.

Nous avons utilisé un programme de test de type « ping-pong » utilisant l'interface de communication « *send/receive* ». Cette interface implique pour chaque message reçu de réveiller le processus en attente afin de traiter le message.

Afin d'évaluer le coût des mécanismes de contrôle d'erreurs, nous avons mené pour chaque type de réseau deux séries de mesures : une série utilisant les mécanismes de contrôle d'erreurs et une série durant laquelle les mécanismes de contrôle d'erreurs ont été désactivés.

Le résultat de ces mesures est présenté dans la suite de ce paragraphe.

E.2.1 Ethernet 100

Nous avons effectué une série de mesures sur un réseau de type Ethernet 100 Mb avec une grappe composée de PCs équipés de processeurs Pentium Pro 200 MHz. Les débits obtenus avec GIMLI sur ce réseau sont présentés sur la figures E.4. La latence de transfert des données en fonction de la taille des messages est présentée dans le tableau E.1.

On peut constater sur la figure E.4 qu'à partir d'une taille de messages de 64 Ko, le débit obtenu est d'environ 12 Mo/s, ce qui correspond au débit maximum physique du réseau. On constate également que les mécanismes de contrôle d'erreurs influencent peu le débit. En effet, la diminution de la bande passante induite par ces mécanismes est comprise entre 5 et 10%. La diminution de la bande passante est due au surcoût induit par le calcul du CRC, réalisé lors de l'émission et de la réception. Ce coût est proportionnel à la taille du message.

La latence d'émission (voir tableau E.1) est de 72 μs avec les mécanismes de contrôle d'erreur et de 50 μs sans contrôle d'erreurs. On constate que la latence est fortement

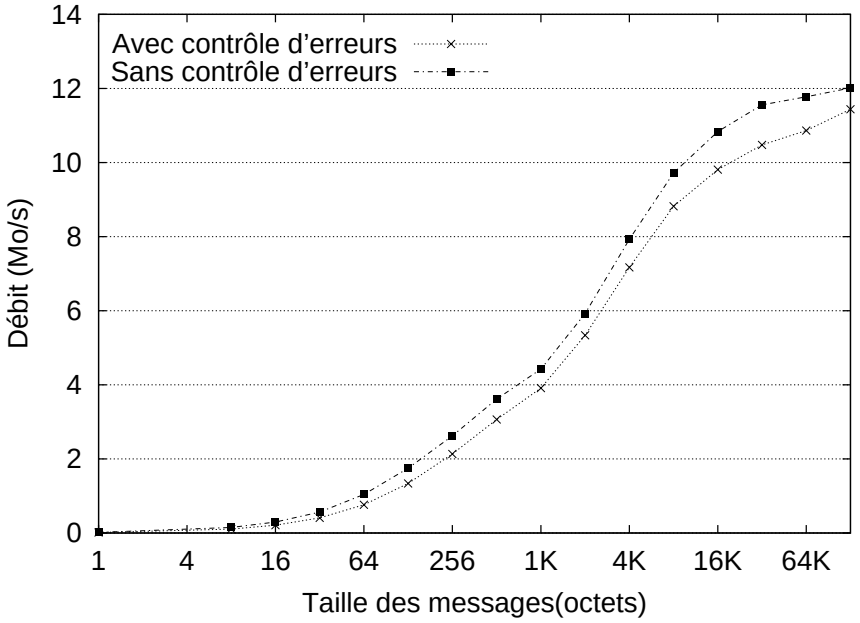

FIG. E.4 – *Débit sur réseau Ethernet 100 / Pentium Pro 200 MHz*

influencée par les mécanismes de contrôle d'erreurs puisque le surcoût induit est de l'ordre de 50%. Ce surcoût est dû à l'attente systématique d'un acquittement lors d'un transfert réseau. Cet acquittement, qui implique l'envoi d'un message sur le réseau, double la latence due au réseau. En observant la différence de latence sur un message de taille nulle entre l'envoi avec et sans contrôle d'erreur, on peut déduire la latence intrinsèque de transmission, à savoir 22 μs. La latence d'émission d'un message de taille nulle avec contrôle d'erreur est donc de 22 μs pour l'envoi du message, 22 μs pour l'envoie de l'acquittement et 28 μs pour la réactivation du processus en attente.

E.2.2 Gigabit Ethernet

Nous avons effectué une série de mesure sur réseau Gigabit Ethernet avec une grappe composée de PCs équipés de processeurs Pentium III 500 MHz. Les débits obtenus avec GIMLI sur ce réseau sont présentés sur la figures E.5. La latence de transfert des données en fonction de la taille des messages est présenté dans le tableau E.2.

On peut constater sur la figure E.5 qu'à partir d'une taille de messages de 64 Ko, le débit obtenu est d'environ 55 Mo/s avec contrôle d'erreurs et d'environ 78 Mo/s sans contrôle d'erreurs. Le débit obtenu est loin du débit physique théorique du support, qui est de l'ordre de 125 Mo/s. On peut également constater que les mécanismes de contrôle d'erreurs ont une influence importante sur le débit, puisque la diminution de la bande passante peut aller jusqu'à 50% dans le cas d'un message de 4 Ko.

La latence d'émission (voir tableau E.2) est de 130 μs avec les mécanismes de contrôle

Taille	Avec contrôle d'erreurs	Sans contrôle d'erreur
0	72	50
1	73	50
8	73	51
16	74	51
64	82	58
256	119	93
1K	263	221
4K	582	496
16K	1705	1454
64K	6169	5235

TAB. E.1 – Temps d'émission (en μs) en fonction de la taille des messages (en octets) sur réseau Ethernet 100 Mb et processeur Pentium Pro 200 MHz

Taille	Avec contrôle d'erreurs	Sans contrôle d'erreur
0	130	19
1	130	20
8	130	22
16	130	23
64	130	27
256	130	40
1K	126	85
4K	286	133
16K	405	413
64K	1217	893

TAB. E.2 – Temps d'émission (en μs) en fonction de la taille des messages (en octets) sur réseau Gigabit Ethernet et processeur Pentium III 500 MHz

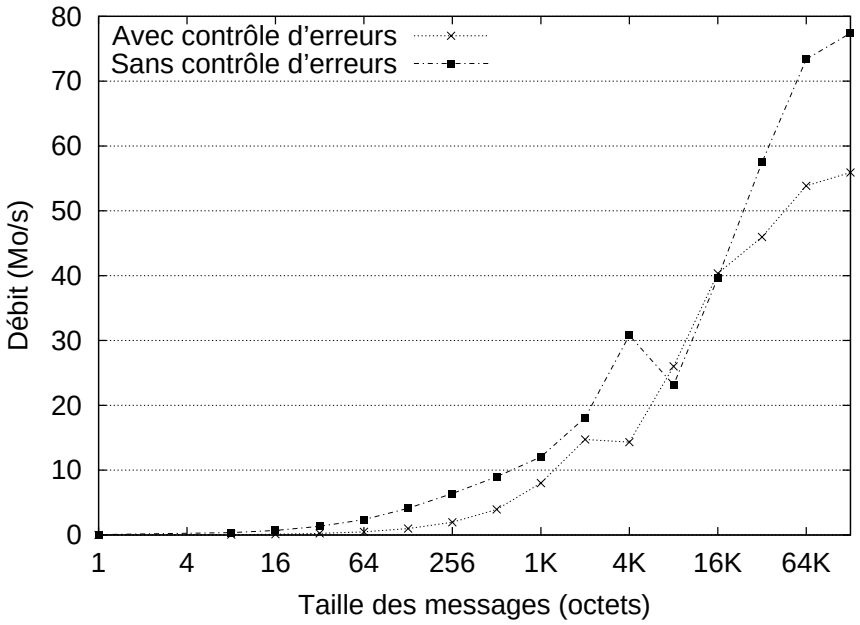

FIG. E.5 – *Débit sur réseau Gigabit Ethernet / Pentium III 500 MHz*

d'erreur et de 19 μs sans contrôle d'erreurs. Nous avons exceptionnellement observé des latences de l'ordre de 28 μs avec l'utilisation du mécanisme de contrôle d'erreurs. Cette mesure exceptionnelle correspond à la valeur que nous devrions théoriquement obtenir. La différence entre la latence mesurée (130 μs) et la latence théorique (28 μs) est probablement due à un problème d'activation du processus en attente. Cependant, nous n'avons pour l'instant pas pu définir avec exactitude la cause de ce décalage. Nous avons pu mesurer que la latence intrinsèque de transmission est de 9 μs et que la durée d'activation d'un processus est d'environ 10 μs.

E.2.3 Amélioration des performances

Les performances obtenues avec la bibliothèque de communication GIMLI peuvent être améliorées de manière significative grâce à plusieurs méthodes : (1) l'utilisation de fenêtres d'acquittements plutôt que l'attente synchrone d'acquittements et (2) l'utilisation de messages actifs.

Dans la mise en œuvre actuelle, l'envoi d'un message est suivi de l'attente synchrone d'un acquittement. Cette attente synchrone double la latence induite par la transmission d'un message sur le réseau. L'utilisation de fenêtres d'acquittements consiste à ne pas attendre l'acquittement de manière synchrone, mais asynchrone. Après l'émission d'un message, la fonction d'émission rend la main au processus appelant sans attendre d'acquittement. Si au bout d'un laps de temps défini par la largeur de la fenêtre l'acquittement n'a pas été reçu, le message est considéré comme perdu et retransmis. Si un acquittement

négatif est reçu, le message est également retransmis. Cette solution implique de conserver une copie du message tel qu'il était lors de la première transmission jusqu'à réception de l'acquittement. Sur des réseaux à haut débit comme Gigabit Ethernet ou Myrinet, cette copie peut cependant engendrer une chute de la bande passante. Une solution à ce problème peut être d'utiliser l'une ou l'autre de ces deux solutions en fonction du type de réseau utilisé et de la taille du message à transmettre.

Une deuxième solution pour améliorer la latence est d'utiliser des messages actifs [96]. Les messages actifs permettent de consommer un message dès son arrivée sur un nœud, au sein même de l'interruption réseau déclenchée lors de sa réception. Ceci supprime le coût de réveil du processus, qui peut se révéler élevé si le nœud est soumis à une forte charge en terme de nombre de processus à exécuter.

F MISE EN ŒUVRE DES LIEURS

F.1 Lieurs d'interface

F.1.1 Lieur d'interface de projection

```
/*
 *   Obtention d'une copie sur page non présente.
 */
unsigned long memory_nopage(struct vm_area_struct *vma,
                            unsigned long address, int write_access)
{
  struct page* page_map ;
  physaddr_t pageaddr ;
  container_t *ctnr ;
  pageid_t pageid ;

  ctnr = (container_t *) vma->vm_linker->ctnr ;

  pageid = ADDR_TO_PAGE (vma->vm_start, virtaddr) + vma->vm_linker->offset ;

  pageaddr = ctnr_find_page ( ctnr->ctnrid, pageid ) ;

  if ( write_access )
    pageaddr = ctnr_grab_page ( ctnr->ctnrid, pageid ) ;
  else
    {
      pageaddr = ctnr_find_page ( ctnr->ctnrid, pageid ) ;
      if ( pageaddr == PHYSADDR_NULL )
        pageaddr = ctnr_get_page ( ctnr->ctnrid, pageid ) ;
      else
        {
          page_map = PAGE_STRUCT(pageaddr);
          atomic_inc(&page_map->count) ;
        }
    }
  return pageaddr ;
}

/*
 *   Obtention d'une copie sur écriture.
 */
unsigned long memory_wppage(struct vm_area_struct *vma,
                            unsigned long virtaddr,
                            unsigned long physaddr)
{
  container_t *ctnr ;
  pageid_t pageid ;

  ctnr = (container_t *) vma->vm_linker->ctnr ;
```

```
  pageid = ADDR_TO_PAGE (vma->vm_start, virtaddr) + vma->vm_linker->offset ;

  return ctnr_grab_page ( ctnr->ctnrid, pageid ) ;
}

/*
 *   Change les droits d'accès à une page virtuelle.
 */
void memory_change_access (void *link, pageid_t pageid,
                           page_state_t state)
{
  vm_linker_data_t *linker = link ;
  virtaddr_t addr ;

  addr = PAGE_TO_ADDR(linker->vma->vm_start, pageid) ;

  switch (state) {
    case INVALID :
      page_change_protection(linker->vma, addr, INVALID ) ;
      linker->vma->vm_mm->rss--;
      break ;

    case READONLY :
      page_change_protection(linker->vma, addr, READONLY ) ;
      break ;

    case READWRITE :
      page_change_protection(linker->vma, addr, READWRITE ) ;
      break ;
  }
}
```

F.2 Lieurs d'entrée/sortie

F.2.1 Lieur d'entrée/sortie en mémoire

```
/*
 *   Création d'une page sur premier accès.
 */
physaddr_t memory_first_touch (container_t *ctnr, pageid_t pageid)
{
  return get_free_page(GFP_KERNEL);
}

/*
 *   Invalidation d'une page.
 */
void memory_invalidate_page (container_t *ctnr, pageid_t pageid)
{
  ctnrPage_t *pageEntry ;

  pageid += ctnr->offset ;
  pageEntry = GET_PAGE_ENTRY ( ctnr, pageid ) ;

  free_page(pageEntry->physAddr) ;
}

/*
 *   Libération d'une page.
 */
void memory_free_page (container_t *ctnr, pageid_t pageid)
{
  ctnrPage_t *pageEntry ;

  pageid += ctnr->offset ;
  pageEntry = GET_PAGE_ENTRY ( ctnr, pageid ) ;

  free_page(pageEntry->physAddr) ;
}

/*
 *   Remplacement d'une page
 */
void memory_flush_page (container_t *ctnr, pageid_t pageid)
{
  /* Non mis en oeuvre */
}
```

F.2.2 Lieur d'entrée/sortie sur fichier

```
/*
 *   Création d'une page sur premier accès.
 */
physaddr_t file_first_touch ( container_t *ctnr, pageid_t pageid )
{
  file_linker_data_t *file_data = ctnr->iolinker_data ;
  struct file *linked_file = file_data->file ;
  struct inode *inode = linked_file->f_dentry->d_inode;
  struct page *page, **hash ;
  physaddr_t addr ;
  size_t offset ;

  pageid += ctnr->offset ;

  offset = pageid * PAGE_SIZE ;

  /*** Teste si la page est présente dans le cache de fichiers ***/

  page = find_page(inode, offset) ;

  if (!page)
    {
      /*** La page n'est pas dans le cache. Chargement depuis le disque ***/

      addr = page_cache_alloc() ;
      page = page_cache_entry(addr) ;
      hash = page_hash(inode, offset) ;
      add_to_page_cache(page, inode, offset, hash) ;

      inode->i_op->readpage(linked_file, page) ;   /*** Chargement ***/
    }

  wait_on_page(page);         /*** Attend la fin du chargement ***/
  addr = page_address(page);

  return addr ;
}

/*
 *   Libération d'une page.
 */
void file_free_page (container_t *ctnr, pageid_t pageid)
{
```

```
  file_linker_data_t *file_data = ctnr->iolinker_data ;
  struct inode *inode = file_data->dentry->d_inode ;
  struct page *page ;
  size_t offset ;

  pageid += ctnr->offset ;
  offset = pageid * PAGE_SIZE ;

  /*** Cherche l'entrée correspondante dans le cache de fichiers ***/

  page = find_page(inode, offset) ;

  if (PageLocked(page))
    __wait_on_page(page);

  page_cache_release(page); /* Decremente le compt. incrementé par Find_Page */
  page_cache_release(page); /* Libére l'entrée du cache de fichier */
}

/*
 *   Libération d'une page.
 */
void file_invalidate_page (container_t *ctnr, pageid_t pageid)
{
  file_linker_data_t *file_data = ctnr->iolinker_data ;
  struct dentry *dentry = file_data->dentry ;
  ctnrPage_t *pageEntry ;
  struct inode *inode ;
  struct page *page ;
  size_t offset ;

  pageid += ctnr->offset ;

  if ( ctnr->linked_node == gobelins_node_id )

    { /*** Supprime la page du cache de fichier ***/

      inode = dentry->d_inode ;
      offset = pageid * PAGE_SIZE ;

      page = find_page(inode, offset);

      if (PageLocked(page))
         __wait_on_page(page);
```

```
      page_cache_release(page);
      page_cache_release(page);

      remove_page_from_hash_queue(page);
      remove_page_from_inode_queue(page);
      page_cache_release(page);
    }
  else
    { /*** Supprime la page de la mémoire locale ***/

      pageEntry = GET_PAGE_ENTRY ( ctnr, pageid ) ;
      page = mem_map + MAP_NR(pageEntry->physAddr);
      __free_page(page);
    }
}

/*
 *   Remplacement d'une page
 */
void file_flush_page (container_t *ctnr, pageid_t pageid)
{
  /* Non mis en oeuvre */
}
```

VU :
Le Directeur de Thèse :

VU :
Le Responsable de l'École Doctorale :

VU pour autorisation de soutenance
Rennes, le
Le Président de l'Université de Rennes 1

Patrick Navatte

VU après soutenance pour autorisation de publication :
Le Président du Jury,

Résumé

Depuis l'apparition des calculateurs parallèles, un effort de recherche important a été réalisé afin de simplifier l'utilisation de ces machines, notamment par l'utilisation d'un modèle de programmation par mémoire partagé et par une meilleure gestion des ressources distribuées. De nos jours, les grappes d'ordinateurs représentent une nouvelle classe d'architecture pour le calcul haute performance bénéficiant d'un rapport prix/performance beaucoup plus élevé que les calculateurs parallèles traditionnels. Cependant, les grappes d'ordinateurs représentent un bond technologique en arrière de par leur architecture totalement distribuée et le manque de système d'exploitation dédié.

Les travaux présentés dans cette thèse portent sur la conception d'un système d'exploitation dédié aux grappes d'ordinateurs fondé sur une gestion globale de la mémoire physique. L'objectif étant de fournir l'illusion d'une machine unique à haute performance au dessus d'une grappe : un **système à image unique**.

Dans un premier temps, nous avons passé en revue les différentes solutions proposées afin de simplifier l'utilisation des grappes d'ordinateurs. Ces solutions, tels que les mémoires virtuelles partagées, les systèmes de gestion de fichiers parallèle et distribué ou la migration de processus, reposent sur des mécanismes de gestion globale des ressources offrant la vision d'une ressource unique au dessus d'un ensemble de ressources distribuées.

Afin d'intégrer l'ensemble de ses services systèmes au sein d'un même système d'exploitation, nous proposons un mécanisme logiciel appelé **conteneur**, fondé sur une gestion de la mémoire physique des nœuds d'une grappe de calculateurs. Ce mécanisme permet de stocker et de partager des données entre des noyaux d'un système d'exploitation hôte s'exécutant sur différents nœuds de la grappe. Les conteneurs sont intégrés au sein du système d'exploitation hôte grâce à un ensemble de **lieurs**. Les lieurs sont des éléments logiciels intercalés à certains niveaux du noyau entre les gestionnaires de périphériques et les services systèmes afin de détourner la gestion des périphériques vers les conteneurs.

Nous démontrons comment l'utilisation conjointe des conteneurs et des lieurs permet de mettre en œuvre de manière très simple des services systèmes tels qu'une mémoire virtuelle partagée, un mécanisme de projection de fichiers en mémoire, un cache de fichiers coopératif et un système de gestion de fichiers distribué. Nous démontrons également comment les conteneurs permettent de simplifier de manière significative les mécanismes de migration de processus, tout en offrant de nouveaux services jusqu'alors inaccessibles.

Un système d'exploitation nommé **Gobelins** a été réalisé sur la base d'un système hôte Linux afin de valider le concept de conteneur. Ce système s'exécute sur une grappe de PCs et intègre des mécanismes de mémoire virtuelle partagée, de cache coopératif et de migration de processus. Avec ce prototype, nous montrons par l'exemple que grâce aux conteneurs, la réalisation de ces mécanismes est très légère et ne demande que quelques modifications du noyau du système hôte.

Mots clefs

Grappe de calculateurs, système d'exploitation distribué, haute performance, mémoire virtuelle partagée, cache de fichier coopératifs, système de gestion de fichiers distribué, migration de processus.