Support Mémoire Adaptable Pour Serveurs de Données Répartis

Olivier Lobry

UNIVERSITE JOSEPH FOURIER-GRENOBLE 1
SCIENCES & GEOGRAPHIE

THESE

pour obtenir le grade de
DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER

Discipline : INFORMATIQUE

Présentée et soutenue publiquement
par

LOBRY Olivier

Le 23 octobre 2000

Support mémoire adaptable pour serveurs de données répartis

Directeur de Thèse
Mme COLLET, Christine

COMPOSITION DU JURY :

M. BALTER, Roland	Président du Jury
M. PUCHERAL, Philippe	Rapporteur
M. FERRIÉ, Jean	Rapporteur
Mme PÉREZ-CORTÈS, Elizabeth	Examinateur
M. DÉCHAMBOUX, Pascal	Examinateur
Mme COLLET, Christine	Directeur de Thèse

Je tiens à remercier l'ensemble des personnes qui par leurs conseils, leurs encouragements ou leur aide ont contribuées à l'aboutissement de ce travail :

Roland BALTER, *Professeur à l'Université Joseph Fourier de Grenoble et responsable du projet Sirac, pour m'avoir fait l'honneur de présider ce jury ;*

Jean FERRIÉ, *Professeur à l'Université de Montpellier, et Philippe Pucheral, Professeur à l'Université de Versailles/St Quentin, et tous deux membres actifs du groupement de recherche OrcTrans, pour avoir accepté de rapporter sur ma thèse ;*

Christine COLLET, *Professeur à l'Institut National Polytechnique de Grenoble, pour avoir assuré la direction de cette thèse pendant ces quatre longues années et supporter mes états d'âmes et humeurs, qui, je dois bien l'admettre, furent parfois un peu moroses ;*

Je n'oublierai certainement pas Pascal DÉCHAMBOUX, *Ingénieur de Recherche à France Télécom R&D, pour avoir co-encadré le début de cette thèse. À lui ainsi qu'à Thierry* JACQUIN, *Ingénieur de Recherche à Rank Xerox, j'adresse de sincères remerciements notamment pour avoir cru jusqu'au bout en l'intérêt de notre effort dans la réalisation du système Arias (paix à son âme).*

Je remercie également mes collègues de bureau, à savoir Philippe, Héléna, Pierre et particulièrement Eliz, pour les riches discutions que nous avons pu avoir. Un grand merci à Genoveva pour m'avoir aidé dans cette dernière ligne droite.

Certaines personnes s'étonneront peut-être de ne pas être mentionnées ou de l'être si peu. À ces personnes je dirais simplement que ceci n'est que le mémoire de ma thèse et non pas celui de ma vie.

Je dédie cette thèse à M.-J., toujours à mes côtés pour calmer mes angoisses, pour m'avoir insufflé mes plus belles et plus riches idées.

Table des matières

Chapitre 1

Introduction

1.1 Problématique et motivation

Les serveurs de données jouent depuis longtemps un rôle central dans la gestion des systèmes d'information. En offrant des services d'interrogation et de manipulation de données et en garantissant un ensemble de propriétés quant à l'utilisation de ces services, ils servent d'intermédiaire entre les applications et les ressources physiques et logiques. Les services mis en oeuvre reposent sur un gestionnaire mémoire, module indispensable du serveur, qui offre une interface permettant le traitement de données permanentes et garantit des propriétés d'exécution et d'intégrité des accès aux données. Les fonctions qu'il implante sont de plus critiques du fait que leur coût d'utilisation établit une borne inférieure au coût d'utilisation des services offerts.

Depuis de nombreuses années déjà, des techniques évoluées ont été étudiées et implantées dans des Systèmes de Gestion de Bases de Données (SGBD). Ces systèmes nécessitent en effet la conception de serveurs capables de prendre en compte des contraintes de gestion très strictes afin de garantir les propriétés ACID des transactions. De nombreuses sociétés se sont consacrées à la construction de tels serveurs capables de garantir ces propriétés tout en supportant un grand nombre d'utilisateurs et de gros volumes d'information.

Avec l'émergence du web, de nouveaux besoins sont apparus conduisant à des services de plus en plus complexes et variés. La taille du réseau mondial augmentant chaque jour, les exigences en termes de nombre d'utilisateurs et volumes de données ont totalement explosé. Dans ce nouveau contexte où les besoins ne sont pas toujours connus à l'avance, la capacité d'un serveur de s'adapter rapidement et à moindre coût aux nouvelles exigences est une propriété vitale que respectent peu ou pas les solutions actuelles. Avec la mondialisation du réseau, la concurrence est bien plus

présente qu'auparavant et les utilisateurs de plus en plus exigeants. L'indisponibilité où la baisse des performances d'un service peut avoir des répercutions directes et immédiates en termes de profits et de parts de marchés.

Bien sûr, face à ces exigences, les fournisseurs de plate-formes d'exécution se sont appliqués à fournir des architectures matérielles "scalables" capables d'évoluer au fur et à mesure que les besoins de capacité d'exécution et de stockage augmentent. Les grappes de machines constituent un exemple incontournable de telles architectures du fait de leur grande capacité d'extension et du faible coût de leur extension. Malheureusement, une architecture scalable permet certes d'absorber une hausse du volume de traitement, mais ne permet pas de prendre en compte de nouvelles contraintes ou comportements. D'autres techniques logicielles doivent donc être étudiées afin de pallier ces lacunes.

1.2 Objectifs et contribution

Ce document a pour objectif de montrer qu'il est fort heureusement possible d'offrir une architecture logicielle permettant de réduire le temps et le coût de développement d'un serveur et de le faire évoluer au fur et à mesure de nouvelles contraintes sans sacrifier totalement sa disponibilité.

Nous proposons le support mémoire adaptable ADAMS (pour ADAptable Memory Support). Sa conception s'appuie sur la constatation qu'il est préférable de bien dissocier les fonctions du gestionnaire mémoire. D'une part, des mécanismes de gestion dont le comportement ne dépend pas des spécificités du serveur doivent permettre l'accès et la modification de l'état du système. D'autre part, des politiques de gestion dont cette fois le comportement dépend des qualités de services et objectifs de performance spécifiques au serveur, doivent prendre des décisions en réaction à des changements d'état du système et modifier cet état en conséquence.

Nous nous plaçons dans le contexte des grappes de machines offrant des possibilités et contraintes de gestion d'une mémoire répartie. Le support ADAMS s'appuie sur un modèle de gestion hiérarchique d'une mémoire répartie. Il se présente sous la forme d'une infrastructure offrant un ensemble de mécanismes de base et permettant l'ajout et le retrait dynamique de politiques de gestion lorsque les besoins se font sentir. En se basant sur un modèle de communication à base d'événement, il facilite l'intégration des politiques tout en séparant bien leurs rôles respectifs. La hiérarchisation de la gestion facilite de plus la cohabitation de différentes politiques répondant à différents besoins ou comportement des traitements.

Afin de bien comprendre l'intérêt de notre proposition il nous semble indispensable que le lecteur ait une bonne compréhension des principes et possibilités de gestion d'une mémoire répartie. En conséquence le but de ce document est double :

- il présente un état de l'art des principes de gestion mémoire et des principales propositions que l'on peut trouver dans la littérature ;
- il propose une architecture pour le support ADAMS et une implantation dans le contexte d'une mémoire virtuelle répartie partagée et permanente.

1.3 Organisation du document

Le document est organisé en huit chapitres dont cette introduction constitue le premier. Les trois chapitres suivants ont pour objectif d'apporter au lecteur une compréhension de la problématique de la gestion d'une mémoire répartie et de lui montrer la variété des propositions faites dans ce domaine. Le chapitre 5 présente un état de l'art des supports et des systèmes adaptables et sert de chapitre "charnière" entre les chapitres précédents et les chapitres suivants dans lesquels nous présentons notre proposition.

Chapitre 2 Ce chapitre présente les caractéristiques d'un serveur de données et plus particulièrement son architecture et ses responsabilités en terme de performance et qualité de service. Nous détaillons un peu plus les tendances d'exploitation des serveurs qui argumentent pour leur capacité d'évolution tant en terme de performance que de qualité de service. Elles justifient notamment le contexte des architectures en grappes de machines dans lequel nous nous plaçons. Nous décrivons ensuite le modèle d'exécution que nous considérons. Celui-ci se compose de l'*espace logique* représentant les données manipulables, de l'*espace de stockage* où peuvent être conservées des images permanentes des données, de l'*espace physique* dans lequel ces données sont manipulées et l'*espace des traitements* regroupant les entités actives qui manipulent les données. Enfin, nous présentons le *gestionnaire de mémoire répartie* (GMR) — le module du serveur qui nous intéresse particulièrement dans ce document.

Chapitre 3 Ce chapitre s'intéresse à la gestion des copies en mémoire répartie en faisant l'hypothèse que les différents traitements n'ont pas de spécificité et peuvent être vus comme un seul traitement réparti sur l'ensemble des machines. Dans ce contexte, l'objectif du GMR est de minimiser en nombre et en coût le défaut d'une donnée en mémoire afin que l'ordonnancement des différents traitements soit le plus proche possible d'un ordonnancement optimal. Trois techniques ont cet objectif en

commun : la prédiction des accès, l'anticipation du chargement des données et l'utilisation de la mémoire distante. Ces techniques sont à la base des politiques de remplacement et de préchargement dont nous présentons les principes et principales propositions dans un contexte centralisé comme dans un contexte réparti. Nous montrons en particulier l'influence du comportement des traitements et concluons par le fait qu'il n'existe pas de politique meilleure que toutes les autres dans tous les contextes d'exécution.

Chapitre 4 Ce chapitre se concentre quant à lui sur la gestion du partage. Cette fois, les traitements sont considérés comme étant concurrents, c'est-à-dire ayant des comportements et besoins différents et non forcément coopératifs. Le GMR est alors responsable de la gestion du partage de l'espace logique et de l'espace physique. La gestion explicite du partage de l'espace physique a comme principal avantage d'isoler les traitements les uns des autres et de permettre aux traitements de s'informer sur l'espace d'allocation afin de s'adapter en conséquence. Les principes de mise en oeuvre de politiques d'allocation, dont l'objectif est la gestion de ce partage, sont présentés puis les principales propositions rencontrées dans la littérature sont détaillées. Nous abordons ensuite le problème du partage de l'espace logique qui se pose lorsque l'accès simultané à une ou plusieurs données par différents traitements peuvent corrompre l'intégrité du système. La notion de modèle de cohérence est présenté ainsi que la nécessité de mettre en oeuvre une politique de cohérence et synchronisation (C&S). Nous nous concentrons sur la notion de sérialisation permettant d'offrir de fortes garanties de cohérence. Différentes techniques de sérialisation sont présentées dans un contexte centralisé et réparti, et utilisant les techniques de versionnement. Nous parlons enfin de l'impact de la sérialisation sur les performances du système et d'autres qualités de service. Nous montrons en particulier comment certaines techniques de C&S permettent de prendre en compte des contraintes de type temps-réel et les différents degrés d'isolation offrant des garanties moins fortes que la sérialisation mais permettant d'atteindre de meilleures performances. La conclusion de ce chapitre, comme celui précédent, est qu'il n'existe pas de politiques meilleures que toutes les autres mais que chacune est adaptée à un domaine d'application bien particulier.

Chapitre 5 Ce chapitre présente l'intérêt d'offrir un support pour faciliter la construction de serveurs de données et son apport en termes de réutilisation et factorisation du code et capacité d'évolution. Il montre ensuite l'intérêt de concevoir un support adaptable, c'est-à-dire dont le comportement peut être adapté en fonction des besoins en performance et qualité de service. Nous caractérisons l'interface et les fonctions

internes que doit offrir un système pour être qualifié de support mémoire dans un contexte réparti. Ces fonctions doivent permettre d'agir sur l'état des espaces logique, physique et de stockage et de réaliser la correspondance entre ces trois espaces. Nous présentons ensuite les principes de base de la capacité d'adaptation et de son impact sur la continuité des services offerts et sur la sécurité du système. Nous étudions ensuite les principaux systèmes pouvant a priori servir de support mémoire et nous montrons comment ils répondent aux exigences que nous avons fixées en terme de gestion et d'adaptabilité. Nous constatons que parmi les systèmes existants, peu offrent un niveau de support suffisant notamment dans un contexte réparti et que la capacité d'adaptation, lorsqu'existante, se limite à un type de politique particulier.

Chapitre 6 Motivé par la conclusion du chapitre 5, nous proposons le support mémoire ADAMS. Celui-ci s'appuie sur un modèle de gestion hiérarchique basé sur la notion de contexte. Un contexte permet d'associer un ensemble de politiques de gestion à un ensemble d'accès. L'organisation hiérarchique de ces contextes offre la possibilité de concevoir de manière hiérarchique un gestionnaire mémoire tout en isolant bien les responsabilités de chaque politique de gestion. L'architecture d'ADAMS est ensuite présentée. ADAMS s'appuie sur le modèle de communication de type notification d'événement/réaction et offre par conséquent une interface de notification et une interface de réaction. La première permet aux politiques d'être alertées des changements d'état du système tandis que la deuxième permet d'agir sur cet état. La modification de l'état du système reste toutefois sous contrôle du support dans le but de garantir certaines propriétés d'intégrité. ADAMS met en oeuvre un ensemble de mécanismes pour la gestion des politiques et des événements permettant de définir des types d'événement, d'abonner des politiques à des événements et de les rattacher à des contextes, ainsi que la notification des événements aux différentes politiques concernées. Le support utilise ces mécanismes pour définir et notifier ses propres événements, mécanismes qui restent toutefois suffisamment ouverts pour permettre l'extension du modèle. Le chapitre se termine par une spécification des interfaces de notification et de réaction et montre comment ces interfaces offrent un support pour la gestion d'une mémoire répartie.

Chapitre 7 Ce chapitre commence par une présentation du système Arias sur lequel nous nous sommes appuyés pour implanter le support ADAMS. Arias offre un support pour la construction d'applications réparties sur une grappe de machines et nécessitant la permanence des données qu'elles manipulent. Il offre pour cela une Mémoire Virtuelle Répartie Partagée (MVRP) permanente. L'originalité de ce système

et qu'il ne définit pas de politique de gestion de la mémoire mais laisse cette tâche à des modules de spécialisation qui peuvent être ajoutés et retirés dynamiquement du système. Nous avons augmenté ce système afin d'y intégrer le support mémoire ADAMS. Bien qu'Arias nous impose des contraintes sur le modèle — notamment l'impossibilité de répliquer une page sur une même machine — il offre un ensemble de mécanismes facilitant l'implantation de notre support. Nous présentons ensuite les choix d'implantation des gestionnaires de politiques et d'événements. Ces choix ont pour principal but l'optimisation des fonctions dont le coût est le plus critique vis-à-vis de leur fréquence d'utilisation. Ces choix nous permettent notamment d'implanter la fonction de notification avec un coût proche de celui d'un appel de procédure. Nous montrons finalement comment notre support peut être utilisé pour implanter un serveur de données réparti et comment il permet de changer de politiques de gestion.

Chapitre 8 Nous concluons ce mémoire de thèse en récapitulant l'apport de notre approche, en précisant les limitations de notre approche, en critiquant certains choix effectués et en donnant quelques orientations afin d'améliorer, consolider et étendre notre proposition.

Chapitre 2

Serveurs de données

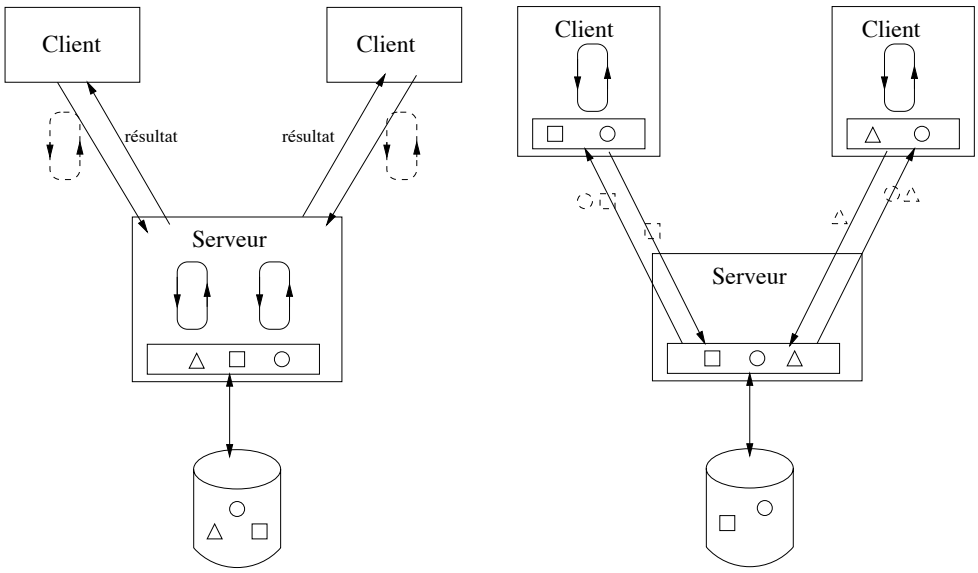
2.1 Caractéristiques des serveurs de données

Les serveurs de données sont à la base des systèmes d'information. Ils peuvent se présenter sous différents aspects et porter différents noms : systèmes de gestion de fichiers, système de gestion de bases de données (SGBD), serveurs Web, entrepôts de données, annuaires, bases de connaissance, etc. Ils ont cependant un objectif en commun qui est de faciliter la manipulation de gros volumes de données stockées sur un support permanent.

2.1.1 Architecture client/serveur

Les serveurs de données offrent un certain nombre de services ayant des caractéristiques communes. Ils permettent tout d'abord d'organiser les données. Ils s'appuient en principe sur un modèle de données (fichier, relationnel, objet, etc.) et permettent la définition et la modification d'une structure homogène liant les données entre elles. Ils mettent en oeuvre des services permettant la manipulation des données. Il peut s'agir de l'ajout de nouvelles données ou de la consultation, de la modification ou de la suppression de données déjà existantes. Les capacités de ces services ainsi que leurs interfaces peuvent cependant fortement varier d'un serveur à l'autre. Le langage déclaratif SQL est un exemple d'interface permettant d'exprimer de manière simple des traitements complexes portant sur de gros volumes de données.

L'interaction entre le serveur et les applications suit généralement le modèle client/serveur. Dans ce modèle, les *processus clients* représentent les utilisateurs des services mis à disposition par le serveur et les *processus serveurs* exécutent ces services. Ces derniers sont entre autre responsables de l'extraction et de la modification



(a) Transfert de traitements (b) Transfert de données

FIG. 2.1 – Architecture Client/Serveur

des données et s'efforcent de garantir un certain nombre de propriétés. La séparation entre processus clients et serveurs permet de protéger ces propriétés du comportement imprévisible des applications auxquelles une confiance seulement limitée peut être attribuée.

La manière dont dialoguent le serveur et ses clients dépend des capacités de traitements de ces derniers. L'approche par *transfert de traitements* consiste à envoyer les traitements vers les données (cf. figure 2.1.(a)). Les clients spécifient les traitements qu'ils envoient au serveur. Ce dernier exécute localement les traitements sur les données de la base puis renvoie les résultats aux clients correspondants. À l'inverse, dans l'approche par *transfert de données*, les clients demandent au serveur de leurs envoyer une copie des données concernées (cf. figure 2.1.(a)). Les données peuvent être récupérées d'un coup ou au fur et à mesure, les traitements sont effectués sur la machine du client puis les modifications sont envoyées au serveur qui les répercute sur la base. Cette deuxième approche augmente le volume de données échangées entre le client et le serveur mais permet de mieux répartir la charge de travail.

Comme nous nous intéressons à la partie serveur, nous ne faisons pas dans ce manuscrit de distinction entre ces deux approches. Dans un cas comme dans l'autre le serveur doit charger en mémoire une copie des données sur lesquelles portent les traitements, modifier ces copies puis les renvoyer dans l'espace de stockage.

2.1.2 Qualité de service

Les serveurs de données assurent généralement un ensemble de propriétés concernant le comportement des services qu'ils proposent. Il en résulte une *qualité de service* (QoS) du point de vue des applications et des contraintes sur l'architecture du serveur et les algorithmes choisis pour implanter les fonctions de base de ce dernier.

Un ensemble de propriétés bien connues dans le monde des bases de données sont la durabilité, l'atomicité et l'isolation des transactions. Ces trois propriétés rassemblées garantissent que l'exécution d'une transaction (ensemble cohérent d'opérations) fait passer l'image permanente de la base d'un état cohérent à un autre état cohérent. Plus exactement, la durabilité apporte une garantie sur la conservation des modifications apportées à la base au-delà de l'arrêt du système. L'atomicité assure que toutes ou aucunes des modifications effectuées par une transaction seront reportées dans la base et permet de prémunir l'état de la base contre d'éventuelles pannes matérielles ou logicielles. Enfin, l'isolation donne l'illusion d'une exécution séquentielle des transactions quand bien même celles-ci sont exécutées simultanément, en parallèle ou en temps partagé.

En plus de ces propriétés, certains Systèmes de Gestion de Bases de Données (SGBD) se fixent des objectifs de QoS supplémentaires. Par exemple, un SGBD multimédia garantit une présentation fluide des vidéos stockées dans la base, quitte à retarder ou ralentir la présentation ou en réduire la définition. Dans un contexte proche, un SGBD temps-réel s'efforce de terminer les transactions dans des délais fixés. Cette garantie peut être forte lorsque le dépassement d'une échéance peut conduire à une catastrophe ou faible lorsque la terminaison avant échéance est une priorité mais n'est pas critique. La haute disponibilité est une autre forme de QoS dont la garantie consiste à assurer une continuité de service. Elle est nécessaire lorsque l'indisponibilité des données peut conduire à l'incapacité de prendre une décision dans une situation critique (ex : surchauffe d'un réacteur nucléaire) ou lorsque l'interruption du service peut conduire à des pertes financières importantes (application boursière).

2.1.3 Performances

Afin de satisfaire pleinement ses utilisateurs, un serveur doit assurer les services qu'il propose et garantir les QoS tout en s'efforçant d'assurer un minimum de performances. Il doit pour cela se fixer un objectif de performance et utiliser efficacement les ressources dont il dispose dans le but d'atteindre cet objectif. Dans le cas d'un service accessible à travers le Web il convient de limiter autant que possible le temps d'attente des usagers afin qu'ils ne s'impatientent et ne finissent par utiliser un service

concurrent. L'objectif de performance pourra dans ce cas être de minimiser le temps de réponse moyen ou bien de minimiser le pire temps de réponse quitte à augmenter le temps de réponse moyen.

Un autre objectif de performance peut être de maximiser le débit du serveur, c'est-à-dire le nombre de traitements qu'il est capable d'effectuer par unité de temps. Cet objectif convient pour un serveur effectuant des traitements courts et non-bloquants dont l'attente du résultat n'est pas critique. Maximiser le débit est nécessaire lorsque les capacités d'absorption sont limitées et que le nombre de requêtes arrivant chaque seconde peut être important. La saturation de la file des requêtes en attente d'activation conduirait à la perte de requêtes, ce qui peut ne pas être tolérable dans certains contextes.

2.1.4 Évolution des serveurs de données

En quelques années, le volume d'information que les serveurs sont capables de traiter a évolué de manière spectaculaire. Il y a à peine cinq ans, Oracle était fier d'annoncer l'utilisation de son SGBD pour des bases d'une centaine de giga-octets. Aujourd'hui, les avancées technologiques permettent de disposer de capacité de traitement et de stockage de plus en plus grandes. Les bases de données ne se mesurent non plus en giga-octets mais en téra-octets (10^{12} octets). La puissance des processeurs actuels rend maintenant possible des calculs et traitements portant sur de gros volumes de données. De nombreux domaines scientifiques tels que la physique des particules, la génétique ou la météorologie ont désormais recours à des serveurs de données capables de gérer des volumes d'information de l'ordre du péta-octet (10^{15} octets).

Dans le domaine des télécommunications, la croissance exponentielle du nombre de téléphones mobiles pose des problèmes de stockage, de traitement et de garantie de qualité de service qui doivent impérativement être résolus sous peine de pertes de parts de marché. De même, dans le domaine de l'audiovisuel, les applications de "vidéo à la demande" doivent pouvoir retrouver puis envoyer de manière fluide des volumes de données importants simultanément à plusieurs terminaux numériques. Ici encore, le nombre d'utilisateurs croît à mesure de la popularité du service et les fournisseurs doivent anticiper et prendre les dispositions nécessaires afin de ne pas être victimes de leur succès.

De plus en plus d'entreprises mettent des services en ligne à disposition des utilisateurs d'Internet. Cette stratégie, qui n'était qu'un atout il y a quelques années, devient de plus en plus une nécessité compte tenu de la concurrence croissante. Les

services de réservation en ligne, les services de consultation et de commandes de produits, les services de commerce électronique ne sont que quelques exemples parmi tant d'autres. Afin de s'approprier des clients ces services doivent être originaux, variés ou personnalisés au profil de leurs utilisateurs. Dans le même temps, le nombre d'utilisateurs et les volumes d'information à traiter croissant chaque jour, les aspects liés aux problèmes de performance ne peuvent être laissés de côté.

Ces quelques exemples montrent que la tendance actuelle est de construire des systèmes d'information capables d'offrir :

- de plus en plus de services de plus en plus complexes
- s'adressant à de plus en plus d'utilisateurs
- et portant sur des volumes de données de plus en plus importants.

Face à ces tendances, les serveurs de données actuels doivent être conçus de manière à pouvoir évoluer rapidement et à moindre frais. Une première approche consiste à s'appuyer sur une plate-forme d'exécution pouvant passer à l'échelle (ou "*scalable*" selon le terme anglais consacré), c'est-à-dire une plate-forme dont la puissance peut être accrue et dont le coût d'extension est proportionnel au gain de performance escompté. Ces plate-formes que nous présentons dans la section suivante permettent d'absorber une hausse du nombre d'utilisateurs et du volume d'information à traiter à moindre frais mais n'aident en rien quant à l'évolution du comportement du serveur. Une deuxième approche, complémentaire à la première consiste à concevoir des serveurs *adaptables*, c'est-à-dire capables d'évoluer d'un point de vue logiciel à des changements du comportement et des besoins des utilisateurs. Nous reviendrons sur ce point dans le chapitre 5.

2.2 Contexte d'étude

Avant d'aborder, dans les chapitres suivants, les principes de gestion d'une mémoire répartie, nous présentons la plate-forme d'exécution sur laquelle nous basons notre étude ainsi que le modèle d'exécution d'un serveur qui nous servira de référence.

2.2.1 Plate-forme d'exécution

La plate-forme d'exécution d'un serveur regroupe l'ensemble des ressources matérielles mises à sa disposition. L'approche centralisée consiste à exécuter le serveur sur une même machine et à partager l'ensemble des ressources par les différents traitements. Afin de limiter le partage du processeur, les serveurs sont généralement exécutés sur des machines parallèles fortement couplées. Ce type d'architecture offre

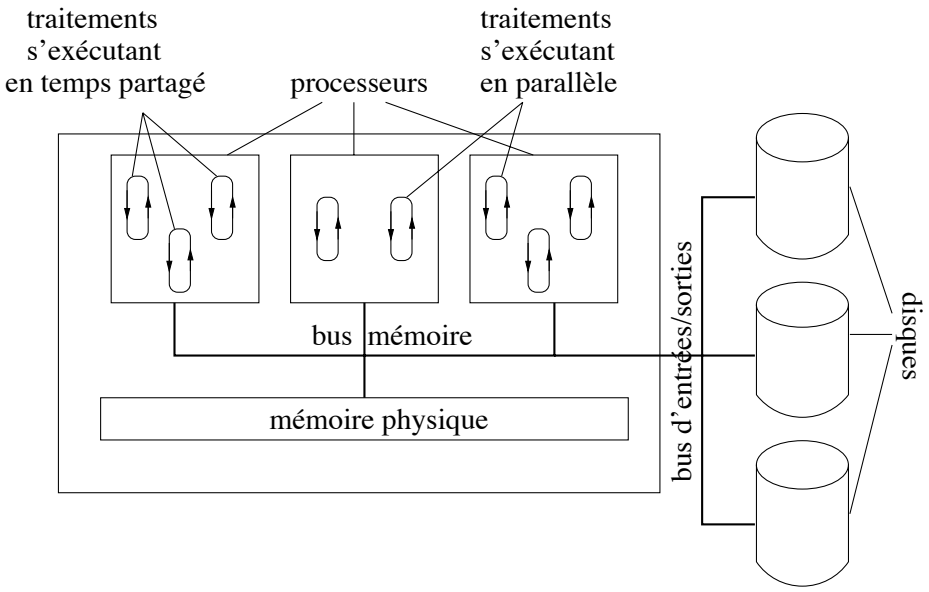


FIG. 2.2 – Architecture parallèle fortement couplée

un grand nombre de processeurs permettant l'exécution simultanée de plusieurs traitements. Cependant, comme le montre la figure 2.2, les processeurs partagent la même mémoire physique et le même bus d'entrées/sorties (E/S).

Ce type d'architecture est notamment utilisée dans le contexte du calcul parallèle, nécessitant généralement peu ou pas d'E/S. Elle est cependant moins adaptée au cas des serveurs de données pour lesquels le partage de la mémoire et du bus d'E/S est bien plus élevé. De plus, le nombre de processeurs que peuvent compter de telles machines reste relativement limité (64 processeurs au maximum) et l'ajout de processeurs et bancs de mémoire est relativement difficile. Enfin, la panne de la machine entraîne l'indisponibilité du serveur et l'inaccessibilité des données.

Une alternative à l'utilisation d'une machine parallèle est de répartir le serveur sur une *grappe de machines* comme le montre la figure 2.3. Dans ce type d'architecture, un ensemble de machines, parallèles ou non, sont reliées par un réseau local. Cette fois-ci deux traitements s'exécutant en parallèle ne partagent pas forcément la même mémoire physique ni le même bus d'E/S. Augmenter la puissance du serveur consiste à ajouter une ou plusieurs machines à la grappe ce qui est plus aisé que l'ajout d'un processeur à une machine parallèle. De plus, le nombre de machines pouvant être interconnectées est significativement plus grand que le nombre de processeurs d'une machine parallèle (il existe aujourd'hui des grappes comptant jusqu'à un millier de machines). Ensuite, l'ajout d'une machine augmente également la capacité de mémoire physique ainsi que le nombre de bus d'E/S. Enfin, ce type d'architecture matérielle

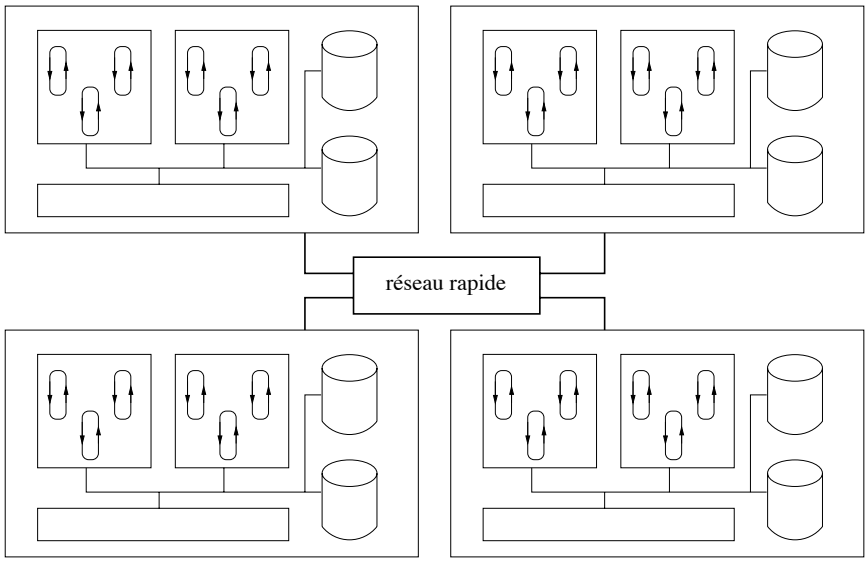


FIG. 2.3 – Architecture en grappe

est bien plus abordable d'un point de vue financier qu'une architecture parallèle. Les machines peuvent être de simples PC dont le prix est relativement bas. Il est de plus possible d'exploiter les ressources des machines de travail déjà existantes et sous-exploitées la plupart du temps.

Les réseaux sur lesquels s'appuie ce type d'architecture doivent offrir une grande bande passante (nombre maximum de bits pouvant être transmis par seconde) et une faible latence (temps minimum pour envoyer un message de quelques octets). Cela est en effet nécessaire du fait que les différentes tâches du serveurs s'exécutant sur des machines différentes peuvent être amenées à partager ou s'échanger des informations. Comme nous le verrons dans le chapitre 3, l'interconnexion des machines permet d'exploiter la mémoire physique de manière globale. Une donnée peut donc être récupérée à partir d'une machine voisine plutôt qu'à partir du disque sur laquelle elle réside.

Notons cependant que contrairement aux croyances, un réseau à grande bande passante n'est pas un avantage en soi car il permet seulement d'offrir un débit équivalent voire légèrement supérieur à celui d'un disque. En effet, les débits des réseaux rapides[1] vont de quelques centaines de Mb/s (ATM, Loopback, HiPPI, SCI) a un Gb/s (Myrinet [TPH]) concurrençant à peine ceux des disques les plus rapides (autour de quelques centaines de Mb/s). Accéder une donnée en mémoire distante à travers un réseau rapide plutôt que sur disque est pourtant avantageux pour deux raisons :

– bien que la latence varie fortement d'un réseau à l'autre (de quelques micro-

[1] Des mesures de performances de la plupart des réseaux rapides peuvent être obtenues à l'adresse *http ://www.netperf.org*.

secondes pour des réseaux à capacité d'adressage tels que SCI ou Myrinet à quelques centaines de micro-secondes pour ATM ou FDDI), elle reste inférieure de plusieurs ordres de grandeur à celle des disques les plus rapides (de 5 à 10 milli-secondes) ;

- les pages étant lues en mémoire primaire (avant d'être envoyées sur le réseau), le temps de lecture d'un ensemble de données est indépendant de leur placement relatif en mémoire distante et de l'ordre de lecture. La lecture de données sur disque nécessite en revanche un processus mécanique relativement long. Le temps de lecture des données dépend donc de l'ordre dans lequel ces dernières sont lues et de leur placement relatif sur le support.

L'architecture en grappe offre de ce fait une plate-forme scalable puisque l'ajout de machines permet de pallier l'augmentation des besoins en ressources matérielles à un coût raisonnable. Notons également que lorsque le réseau supporte la panne d'une machine, celle-ci n'entraîne pas forcément la panne complète du serveur qui peut continuer à s'exécuter en mode temporairement dégradé. Ceci est particulièrement important lorsque la continuité de service est un critère de qualité important. En conséquence, nous considérons dans notre étude le cas les serveurs conçus pour s'exécuter sur une grappe de machines.

2.2.2 Modèle d'exécution

Sauf mention contraire nous supposerons que les machines de la grappe sont monoprocesseur. Chacune d'elles possède une mémoire physique locale, éventuellement un ensemble de disques et elles sont inter-connectées par un réseau local rapide. Cette plate-forme d'exécution est supposée pouvoir accueillir un ou plusieurs serveurs de données. Afin de simplifier la présentation du modèle de référence, nous supposerons cependant qu'un seul serveur s'exécute sur la grappe. Le serveur pourra être constitué de plusieurs sous-serveurs répartis sur les différentes machines de la grappe. Les processus clients et serveurs respectent l'architecture client/serveur traditionnelle. Un client peut s'exécuter soit à l'extérieur de la grappe soit sur une des machines de la grappe et partager des ressources physiques avec le serveur.

Le modèle d'exécution se compose de quatre espaces que nous avons représentés dans la figure 2.4 :

Espace de stockage Le serveur permet la manipulation de données permanentes. Ces données sont stockées de manière durable dans un *espace de stockage* que nous supposons constitué d'un ou plusieurs disques accessibles par les machines de la grappe. La *base de données* regroupe l'ensemble des données permanentes.

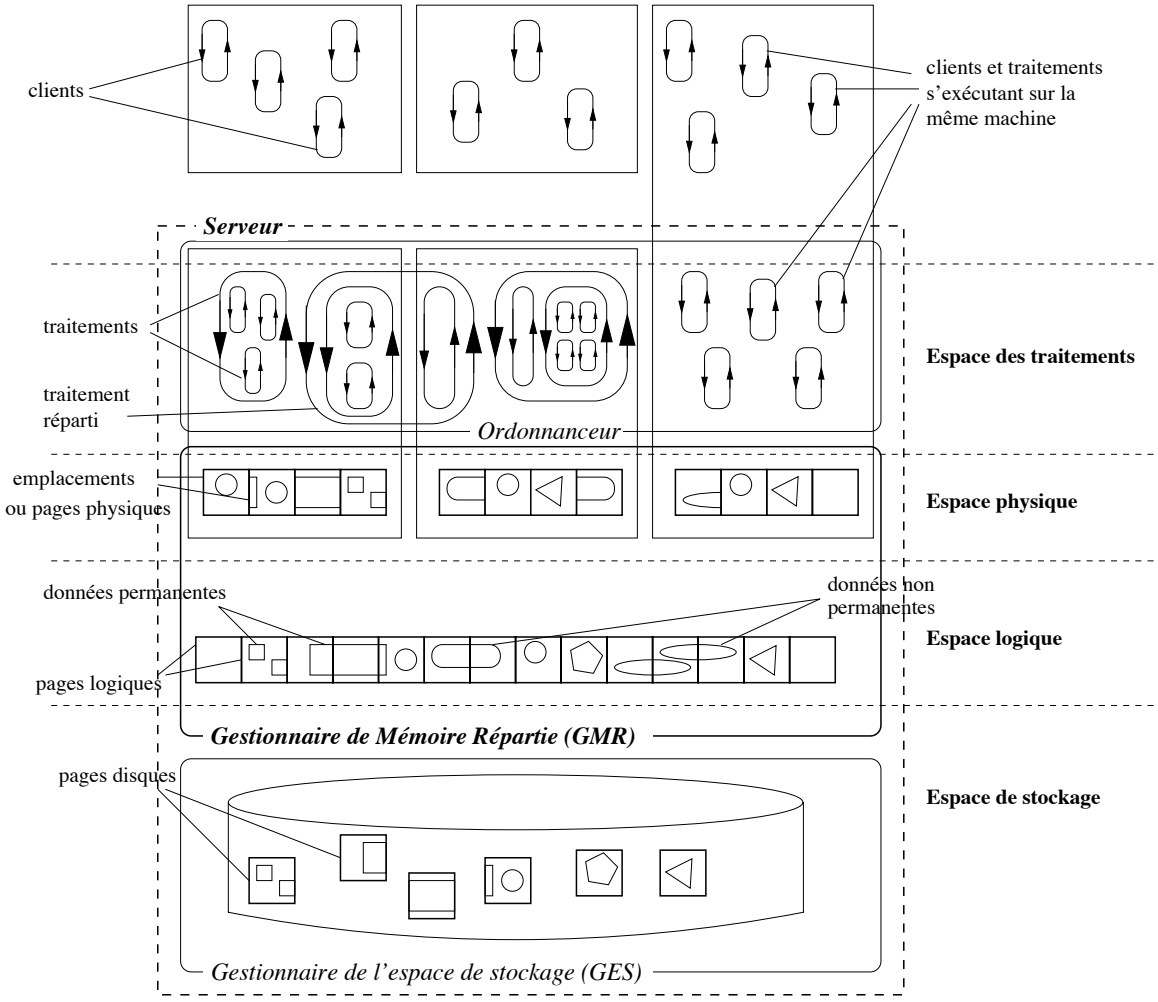


FIG. 2.4 – Modèle d'exécution de référence.

Elle peut être centralisée sur un seul disque, répartie sur plusieurs et éventuellement répliquée. La manière dont elle est organisée ne nous intéresse cependant pas. Nous supposons qu'il existe un *gestionnaire de l'espace de stockage* (GES) dont l'interface permet de lire une donnée depuis l'espace de stockage vers la mémoire physique et, inversement, d'écrire une donnée modifiée depuis la mémoire physique vers l'espace de stockage. Nous supposons également que le coût des lectures et écritures est sensiblement le même quelle que soit la machine depuis laquelle elles sont effectuées. Une donnée dans cet espace est identifiable de manière non ambiguë. De plus, l'espace est découpé en *pages*, unité d'échange entre le disque et la mémoire.

Espace logique L'*espace logique* est un espace d'adressage regroupant toutes les données pouvant être lues et modifiées par le serveur. Une donnée admet un identifiant unique et regroupe un ensemble d'informations ordonnées et sémantiquement liées. Cet espace regroupe non seulement les données permanentes de la base et identifiables par les clients (relations, n-uplets, objet complexe, définition d'une classe, vidéo, image, etc.) mais également des données utiles à l'exécution des traitements ou à la gestion du serveur et non accessibles par les clients (index, tables de hachage, verrous, statistiques, informations de localisation, etc.). Cette espace est divisé en pages, appelées *pages logiques* de la même taille que celles de l'espace de stockage.

Espace des traitements Nous appelons *traitement* toute activité constituée d'un ou plusieurs flots d'exécution, effectuant des opérations sur les données de l'espace logique. Ces opérations peuvent être réalisées soit pour le compte de clients, soit pour la gestion même du serveur. Notons qu'un traitement peut éventuellement regrouper plusieurs traitements. Par exemple, la jointure de deux relations est un traitement faisant partie du traitement de plus haut niveau que représente une requête, elle-même incluse dans une transaction. Un traitement peut également représenter un processus de gestion du serveur tel que celui permettant l'échange de pages ou d'objets entre le serveur et ses clients ou encore un ramasse-miette. Un traitement peut être réparti sur un ensemble de machines mais les flots d'exécutions qui le constituent s'exécutent chacun sur une machine donnée (pas de migration de flots). Un serveur réparti peut donc être vu comme un traitement réparti, constitué de traitements de plus bas niveau. L'*espace des traitements* regroupe bien entendu l'ensemble des traitements s'exécutant sur la grappe. L'ordre dans lequel les traitements doivent être exécutés est décidé par l'*ordonnanceur*. Celui-ci peut à tout moment décider d'activer ou suspendre

un traitement dans le but d'atteindre un objectif de performance visé ou de garantir des temps d'exécution.

Espace physique Sur chaque machine de la grappe, une partie de la mémoire physique est réservée pour le compte du serveur. L'ensemble de ces parties de mémoire constitue l'espace physique. Il est, tout comme l'espace logique et l'espace de stockage, découpé en pages de tailles égales. Ces *pages physiques* sont appelées également *emplacements* et ont la même taille que les pages logiques. Une *copie* désigne l'association d'une page logique et d'une page physique. Une copie a une *valeur* qui peut être différente de celles d'autres copies de la même page. Cette valeur correspond à l'état des octets que constitue la page physique associée.

2.2.3 Gestionnaire de mémoire répartie

Le gestionnaire de la mémoire répartie (GMR) d'un serveur de données joue le rôle de médiateur entre les traitements et les données. C'est lui qui permet aux traitements d'accéder à l'espace logique à travers l'utilisation de l'espace physique. Il a de ce fait pour rôle d'établir et maintenir la correspondance entre les pages logiques et les emplacements et doit pouvoir garantir la validité des identifiants de pages physiques qu'il passe au traitements. Il doit pour cela pouvoir détecter l'accès aux pages soit en imposant aux traitements de déclarer explicitement leurs intentions par le biais d'une interface d'accès, soit pouvoir détecter implicitement l'accès aux pages en s'appuyant sur les mécanismes de détection de défaut de page ou de violation d'accès que le matériel met à sa disposition.

Lorsque des qualités de service sont garanties par le serveur, le GMR peut être le garant d'un certain nombre de propriétés. Il peut pour cela être amené à par exemple interdire temporairement ou définitivement l'accès à une page logique ou bien récupérer des pages physiques allouées pour un traitement. Il doit dans le même temps s'efforcer d'orienter ses décisions de gestion vers l'objectif de performance choisi.

Le GMR met en oeuvre un certain nombre de *politiques de gestion*. Par politique de gestion nous entendons un algorithme capable de réagir à des événements de changement d'état du système et de prendre des décisions de manière à modifier cet état. L'objectif général des politiques est de maintenir l'état du système le plus stable possible. Un état instable est par exemple l'intention d'un traitement actif d'accéder une donnée pour laquelle il n'a aucun droit d'accès. Remettre le système dans un état stable consistera par exemple à lui accorder des droits d'accès, le suspendre ou l'annuler. Le GMR met également en oeuvre un ensemble de *mécanismes*

de gestion permettent aux politiques d'agir sur l'état du système. Les algorithmes sur lesquels reposent ces mécanismes doivent être choisis convenablement afin d'être capable d'atteindre de bonnes performances. Il est notamment primordial que ces algorithmes soient scalables c'est à dire dont le coût en utilisation des ressources (processeur, mémoire, réseau) soit le plus linéaire possible par rapport au nombre d'utilisateurs et au volume de données.

2.3 Conclusion

Face à la croissance incessante du volume d'information et du nombre d'utilisateurs, et face à l'émergence chaque jour de nouveaux besoins, il devient impératif de construire des serveurs capable d'évoluer en terme de puissance et d'architecture. Dans ce chapitre nous avons situé le contexte d'étude du document. Nous supposons que les serveurs s'exécutent sur une grappe de machines compte tenu de la capacité d'évolution d'une telle plate-forme. Le modèle d'exécution sur lequel nous nous basons définit quatre espaces : l'espace de stockage, l'espace de logique, l'espace physique et l'espace des traitements. Dans ce modèle, le gestionnaire de la mémoire répartie (GMR) du serveur doit permettre aux traitements d'accéder, dans l'espace physique, des données identifiées dans l'espace logique. Il met pour cela en oeuvre des politiques et des mécanismes de gestion ayant des rôles bien distincts. Selon les caractéristiques du serveur mis en oeuvre, les politiques qui constituent le GMR peuvent être amenées à garantir certaines propriétés tout en assurant de bonnes performances. Les deux chapitres suivants expliquent les principes de gestion d'une mémoire répartie et les politiques de gestion qui doivent apparaître dans l'architecture du GMR.

Chapitre 3

Gestion de copies en mémoire répartie

3.1 Introduction

3.1.1 Contexte d'étude

Ce chapitre traite de la gestion des copies de pages logiques dans l'espace physique. Une copie est une représentation physique d'une page logique dans la mémoire locale d'une machine de la grappe. L'ensemble des copies définit ainsi une association de l'espace logique vers l'espace physique. Cette association n'est ni injective puisque une page peut ne pas être représentée en mémoire primaire — nous disons dans ce cas que la page est *non résidante* — ni surjective puisqu'une même page peut admettre plusieurs copies — nous disons alors que la page est *répliquée*.

Nous ne faisons ici aucune distinction entre les traitements dans le sens où nous ne nous intéressons pas aux problèmes de partage de l'espace logique ou de l'espace physique liés à l'exécution simultanée des traitements. Le cas où les traitements sont vus comme étant indépendants les uns des autres est traité dans le chapitre suivant. Nous considérons donc que l'ensemble des traitements est regroupé en un seul traitement réparti sur l'ensemble des machines de la grappe. Comme nous l'avons précisé dans le chapitre précédent, les flots d'exécution qui le constituent s'exécutent toujours sur la même machine et accèdent les données uniquement dans la mémoire de leurs machines d'exécution respectives.

Nous supposons en outre que la gestion de la cohérence des réplicas, aspect également abordé dans le chapitre suivant, suit le paradigme "Read One/Write All" qui stipule qu'une page en cours de modification admet une copie unique et que toutes les

copies d'une page en lecture seule ont la même valeur. Cela nous permet de supposer que lorsque l'accès à une page est autorisé, n'importe quelle copie de la page présente dans l'espace physique peut servir à construire une nouvelle copie.

3.1.2 Motivation

Un contexte idéal permettant d'atteindre l'objectif de performance visé est celui où à aucun moment une contrainte matérielle ou logicielle ne peut empêcher l'exécution d'un quelconque traitement. L'ordonnanceur peut décider à loisir de suspendre l'exécution d'un traitement pour en exécuter un autre. Le temps d'exécution d'un traitement est alors égal à la somme des temps d'exécution de chacune de ses instructions. L'objectif de performance peut être atteint à condition d'ordonner convenablement les traitements et de les interrompre lorsque nécessaire. Remarquons que maximiser l'utilisation du processeur ne peut que contribuer à améliorer l'objectif de performance visé quelqu'il soit.

Dans la pratique, les ressources matérielles disponibles ne sont cependant jamais infinies. Cela est notamment le cas de la mémoire physique dont la taille est de plusieurs ordres de grandeurs inférieure à celle de l'espace de stockage. Dans ces conditions, il n'est pas improbable qu'un traitement porte sur une page logique non résidante. Lorsque cela arrive, nous parlons de *défaut de page*. Le traitement est interrompu le temps qu'une entrée/sortie (E/S) soit effectuée afin de charger la page concernée en mémoire physique. Le temps d'exécution d'un traitement est alors égal à la somme des temps d'exécution de chacune de ses instructions plus le temps total attendu sur des défauts de page.

La limitation de la mémoire physique restreint les possibilités d'ordonnancement puisqu'un traitement ne peut être appliqué qu'à des données en mémoire. À tout moment, l'ordonnanceur ne peut choisir d'activer un traitement que si celui-ci n'est pas bloqué en attente du chargement d'une page. Il est donc amené à effectuer un choix (activer un autre traitement) qui n'aurait peut-être pas été le sien si toute la base tenait en mémoire.

3.1.3 Gestionnaire de mémoire répartie

Un des objectifs du Gestionnaire de Mémoire Répartie (GMR) GMR est par conséquent de gérer les copies de manière à ce que les ordonnancements qu'il permet soient les plus proches des ordonnancements optimaux. Il doit pour cela :

– minimiser le nombre de défauts de page afin de minimiser les contraintes de

blocage,
- minimiser le coût (en temps) des défauts de page afin de rendre les traitements ré-activables le plus rapidement possible.

Minimiser le nombre de défauts de page est d'autant plus critique que le temps de chargement peut est grand ce qui est le cas lorsque le gestionnaire de l'espace de stockage est saturé de requêtes. En effet, les demandes d'E/S d'un disque sont traitées de manière séquentielle. Le temps de traitement d'une demande d'E/S dépend donc du nombre d'E/S déjà en attente au moment où elle est initiée. Si la périodicité des demandes est inférieure ou égale à α, la latence du disque, chaque E/S peut être effectuée immédiatement en un temps proche de α. Si en revanche, à cause d'une mauvaise gestion mémoire, la périodicité augmente au point de dépasser la latence du disque, le nombre d'E/S en attente croît jusqu'à ce que tous les traitements se retrouvent bloqués. Le temps de chargement d'une page est alors égal à αn, où n est le nombre de traitements. Dans une telle situation, même lorsque l'objectif de performance est de maximiser l'utilisation du processeur (qui est l'objectif le plus basique car il n'impose aucune contrainte d'ordonnancement), admettre un grand nombre de traitements dans l'espoir qu'il y en ait toujours un d'activable peut être contraire à l'objectif visé.

3.1.4 Organisation du chapitre

Ce chapitre présente les différentes politiques de gestion de copie que le GMR doit mettre en oeuvre. Il commence par donner les principes de gestion des copies dans une mémoire répartie et les moyens dont dispose le GMR pour parvenir à exploiter au mieux les ressources de la plate-forme avant de de détailler les politiques proprement dites.

3.2 Méthodes de gestion

La gestion de copie repose sur la notion d'*utilité* des copies, poids qui leur est donné en fonction de l'intérêt de les conserver en mémoire. L'utilité d'une copie C d'une page logique P sur une machine M dépend essentiellement :
- de la probabilité que la page soit accédée par un traitement,
- du temps de chargement qu'elle permet d'économiser

Par exemple, si la page P contient des données qui ne seront plus jamais accédées, il n'est pas utile de conserver la copie C puisqu'elle occupe inutilement de la place en

mémoire. Si il existe une copie C' de la même page sur une machine M' et que la page P ne peut être accédée que sur la machine M', la copie C n'a là encore aucune utilité.

Bien sûr, l'utilité d'une copie peut dépendre d'autres facteurs. Par exemple pour un serveur temps-réel, l'utilité des copies peut également dépendre de l'ordre de priorité des traitements. Afin d'être le plus indépendant possible de l'objectif de performance visé, nous supposerons que la probabilité d'accéder une page correspond à sa probabilité d'être accédée dans un ordonnancement optimal et par conséquent tenant compte de l'objectif visé.

Le GMR doit donc maximiser l'utilité des copies et possède pour cela trois méthodes : prédire le comportement du traitement global, utiliser la mémoire de manière globale et anticiper le chargement des copies.

3.2.1 Prédiction, remplacement et préchargement

Prédire le comportement d'un traitement permet de prédire les pages qu'il va prochainement accéder et permet donc de calculer l'utilité d'une page. Les techniques de prédiction peuvent être classées dans deux catégories : les techniques de prédictions *conservatrices* et les techniques de prédiction *progressistes*. Ces techniques se distinguent selon les informations qu'elles exploitent et le type d'information qu'elles fournissent au GMR :

Prédictions conservatrices Une prédiction conservatrice est une *prédiction à court terme*. Ce sont généralement des techniques *probabilistes* qui estiment pour l'ensemble des pages résidantes lesquelles ont le moins de chance d'être parmi les prochaines accédées. Ce sont des techniques de prédiction *restreintes* dans la mesure où seules les pages résidantes sont considérées. Nous les qualifions de conservatrices car leur objectif est de *conserver* le système dans le meilleur état possible en épargnant les pages les plus utiles.

Prédictions progressistes Une prédiction progressiste, à l'inverse, tente de voir l'avenir à *moyen* ou *long terme* et de manière plus précise. Les prédictions sont généralement *déduites* d'exécution passées ou d'une analyse algorithmique des traitements, ou tout simplement *avisées* par les programmes. Ces techniques sont *étendues* car portant sur l'ensemble de la base et essaient de prédire *quand* et *avec quelle probabilité* une page va être accédée. Nous les qualifions de progressiste dans la mesure où elles essaient de se projeter dans le futur et d'inciter le système à faire *progresser* son l'état en le prévenant de l'accès à de nouvelles pages.

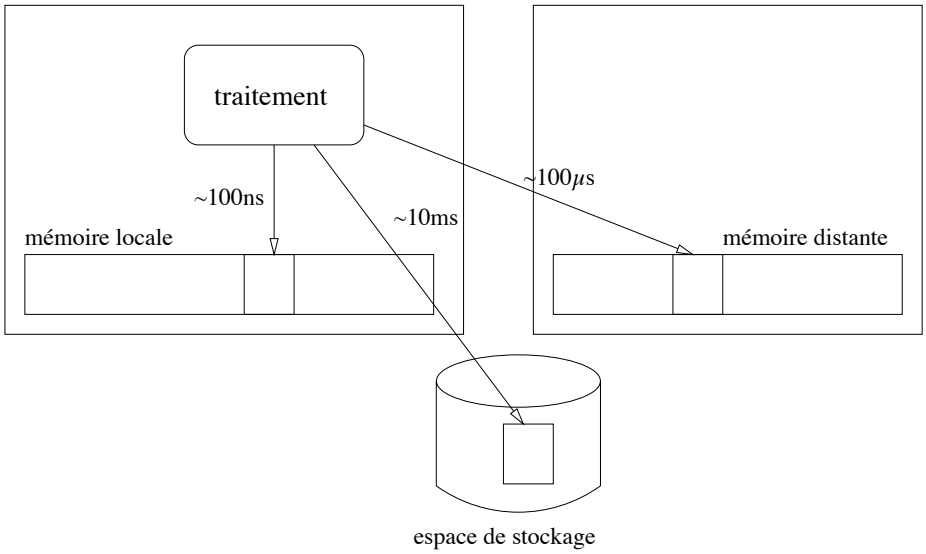


FIG. 3.1 – Trois niveaux de mémoire dans une grappe de machines.

3.2.2 Utilisation de la mémoire distante

Du point de vue d'un flot d'exécution, une grappe de machine offre trois niveaux de mémoire (cf. figure 3.1)

- la *mémoire locale* qui est la mémoire physique de la machine sur laquelle s'exécute le flot,
- la *mémoire réseau* ou *mémoire distante* constituée de l'ensemble des mémoires physiques des autres machines de la grappe,
- et la *mémoire permanente* constituée de l'ensemble des disques rattachés à la grappe.

Ces trois niveaux se distinguent par le temps nécessaire pour y lire une page. Le temps de lecture d'une page en mémoire locale est dominé par la latence de la du bus mémoire et de la mémoire physique et est de l'ordre de $10^{-7}s$. Le temps de lecture d'une copie en mémoire distante est dominé par la latence du réseau et se situe autour de $10^{-4}s$. La lecture d'une copie sur disque nécessite cette fois une E/S impliquant des composants mécaniques et dont le coût est de l'ordre de $10^{-2}s$. Les temps d'accès effectifs aux différents niveaux peuvent varier selon la configuration matérielle de la grappe. Néanmoins, le temps d'accès en mémoire locale reste plusieurs ordres de grandeurs inférieur à celui de la mémoire distante, lui-même plusieurs ordres de grandeurs inférieur à celui du disque.

L'utilité d'une copie est fonction du temps de blocage qu'elle permet d'éviter. Nous appelons *gain d'accès relatif* (GAR) d'une copie C pour une machine M, le temps nécessaire à M pour obtenir une copie locale si C n'existait pas. Le GAR d'une

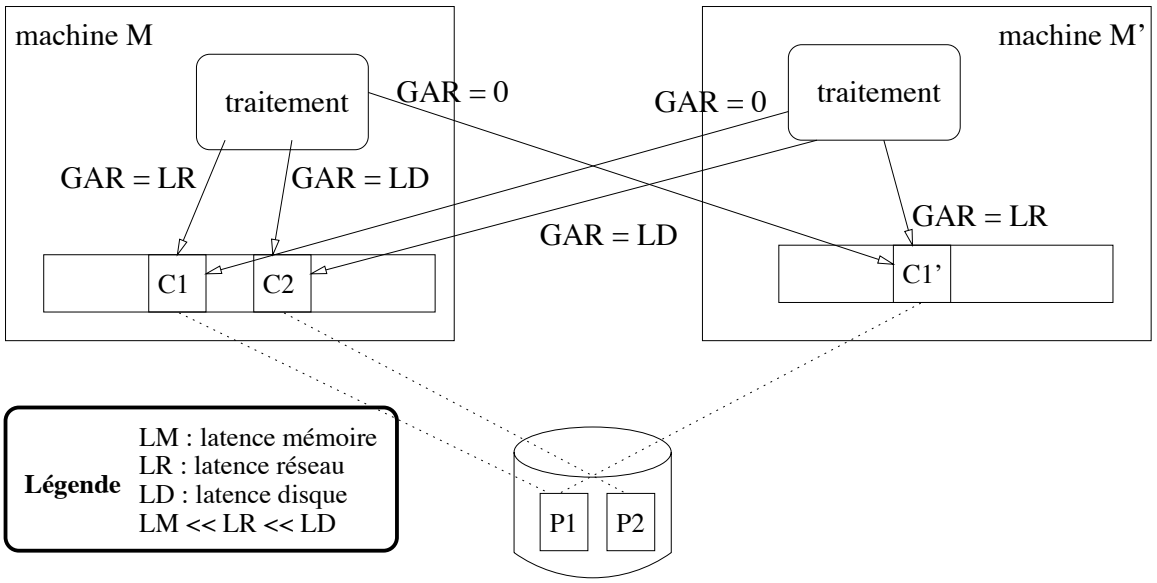


FIG. 3.2 – Gain d'accès relatif (GAR) en mémoire répartie.

copie dépend :

- de la proximité de la copie par rapport aux flots d'exécution susceptibles de l'accéder,
- de la présence d'autres copies de la même page dans la mémoire répartie.

Supposons, comme l'illustre la figure 3.2, deux pages logiques P1 et P2 et une grappe constituée des machines M et M'. P1 admet deux copies C1 et C1' respectivement en mémoire physique des machines M et M'. P2 en revanche n'admet qu'une copie C2 en mémoire physique de M. Du point de vue de M, C1 permet d'éviter de lire C1' tandis que le GAR de C1' est nul. La copie C2, en revanche permet d'éviter une E/S. En supposant que sur M, P1 et P2 ont la même probabilité d'accès, il semble plus intéressant de remplacer P1.

Du point de vue de M', même si l'accès à C2 nécessite cette fois un accès au réseau, elle permet également d'éviter une E/S. De son côté, C1' permet d'éviter de lire la copie C1, laquelle, du point de vue de M', n'a aucune utilité.

D'une manière générale, accéder à la mémoire réseau est profitable dans deux situations qui sont illustrées par la figure 3.3 :

- Lorsqu'une page P est ou a été accédée sur une machine distante, une copie de P peut être obtenue localement à moindre coût. Le GMR peut donc tirer profit du partage réparti des données afin de réduire le coût des défauts de page.
- Lorsque la répartition des traitements entraîne une mauvaise répartition des besoins en mémoire sur l'ensemble de la grappe, certaines mémoires locales

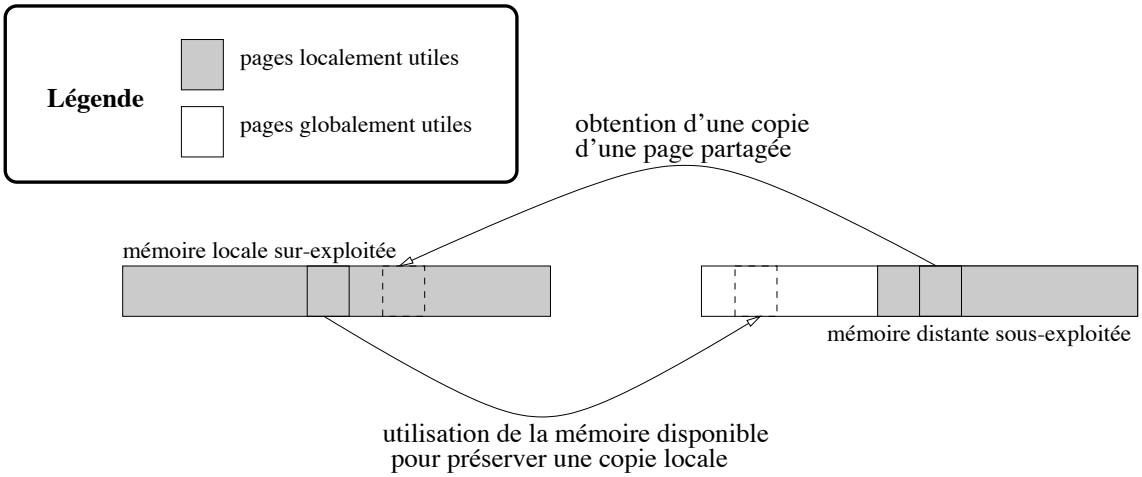


FIG. 3.3 – Utilisation de la mémoire répartie pour réduire le coût des défauts de page.

peuvent être sur-exploitées pendant que d'autres sont sous-exploitées. Les mémoires sous-exploitées peuvent alors être utilisées afin de préserver en mémoire répartie des copies de pages qui ne peuvent être conservées dans les mémoires sur-exploitées. Cela permet de réduire le coût des prochains défauts de ces pages.

3.2.3 Anticipation des accès

Afin de limiter le coût voire le nombre des défauts de page, une méthode consiste à anticiper le chargement des pages non résidantes avant qu'elles ne soient accédées par les traitements. Cette méthode appelée, préchargement de données, est illustrée par la figure 3.4. Lorsque le système n'a pu prévoir l'accès à une page (a), le temps d'attente est maximal. Une anticipation même tardive (b) permet de réduire le temps que passe le traitement à attendre. L'idéal est que la page arrive en mémoire au moment même où elle est accédée (c). En effet, une anticipation trop précoce n'apporte pas de gain supplémentaire (d).

3.2.4 Politiques et mécanismes de gestion

Le GMR a comme objectif de prédire et anticiper l'accès aux pages et de tirer profit du partage de données et du déséquilibre de charge dans le but de réduire les défauts de pages en nombre et en coût. Il s'appuie ou met en oeuvre une technique de prédiction soit conservatrice soit probabiliste.

Une prédiction conservatrice permet d'estimer l'utilité de conserver ou détruire une copie déjà existante. Elles sont pour cela assimilées, comme le montre la figure 3.5 à des

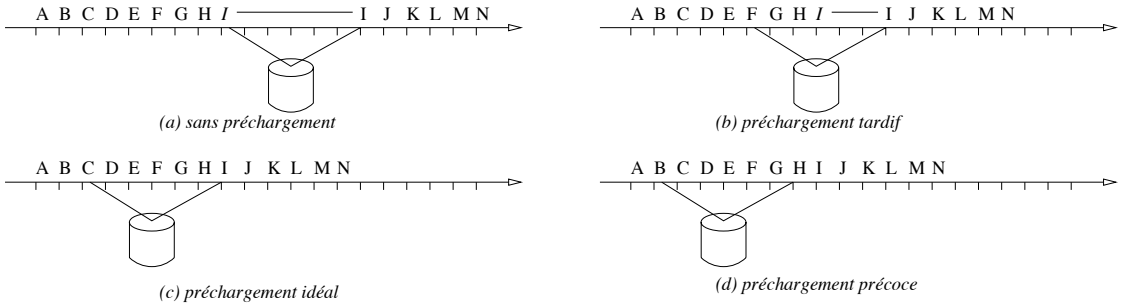


FIG. 3.4 – Préchargement d'une page.

prédiction		action		politique
conservatrice	→	que détruire ou conserver	→	remplacement
	↗			
progressiste	→	que et quand créer	→	préchargement

FIG. 3.5 – Lien entre les techniques de prédiction et les politiques de remplacement et de préchargement.

politiques de remplacement. Une prédiction progressiste permet en revanche d'estimer l'intérêt d'anticiper la création d'une copie et sont pour cela assimilées à des *politiques de préchargement*. Cependant, une prédiction déductive permet également d'estimer l'utilité de conserver ou détruire une copie et peut pour cette raison être utilisée par une politique de remplacement.

Politique de remplacement Une politique de remplacement est nécessaire dès lors que l'ensemble des pages logiques accédées par les traitements est supérieur à la taille de l'espace physique. Lorsque tous les emplacements contiennent déjà une copie et qu'un traitement exprime son intention d'accéder une page non résidante dans sa mémoire locale, il convient de libérer un des emplacements de manière à pouvoir charger la page désirée. Le rôle de la politique de remplacement est de décider quelle page doit être remplacée localement et éventuellement de réorganiser la mémoire répartie afin de maximiser l'utilité des copies résidantes. Traditionnellement, son objectif est de réduire le nombre et sinon le coût des défauts de page. Elle met généralement en oeuvre une technique de prédiction conservatrice mais peut également bénéficier des conseils avisés d'une prédiction progressiste.

Politique de préchargement L'objectif des techniques de préchargement est de réduire le coût et si possible le nombre des défauts de page en anticipant le chargement

des pages qui vont probablement être accédées. Elles se basent sur des techniques de prédiction progressistes qui peuvent faire partie intégrante de la politique ou être implantée à l'extérieur. Dans le deuxième cas, le préchargement peut être contrôlé depuis l'extérieur et le GMR doit fournir une interface permettant d'anticiper la création d'une copie et le chargement des données. Notons que le préchargement ne peut être simplement provoqué en anticipant l'accès aux pages à précharger. En effet, lors d'une tentative d'accès, le GMR peut être amené à prendre des décisions qui ne peuvent ou ne doivent être prises que lorsque le traitement est effectivement prêt à accéder la page.

3.3 Politiques de remplacement

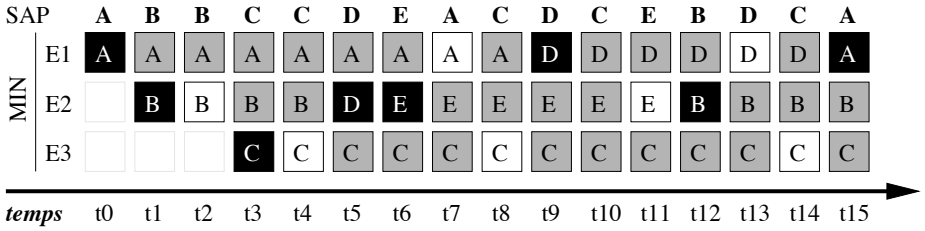
Cette section présente différentes propositions de politiques de remplacement. Dans un soucis de clarté, nous présentons le problème du remplacement en mémoire locale puis en mémoire répartie. Dans le premier cas, la mémoire distante n'est pas utilisée et l'utilité d'une copie dépend uniquement de sa probabilité d'accès. Les politiques proposées en 3.3.1 tentent de conserver les pages qui ont le plus de chance d'être prochainement accédées. Elles s'apparentent pour cela à des techniques de prédiction conservatrices. En revanche, en mémoire répartie, l'utilité d'une copie dépend également de son gain d'accès relatif. Les politiques de remplacement proposées en 3.3.2 s'apparentent de ce fait plutôt à des techniques de gestion de mémoire globale.

3.3.1 Remplacement en mémoire locale

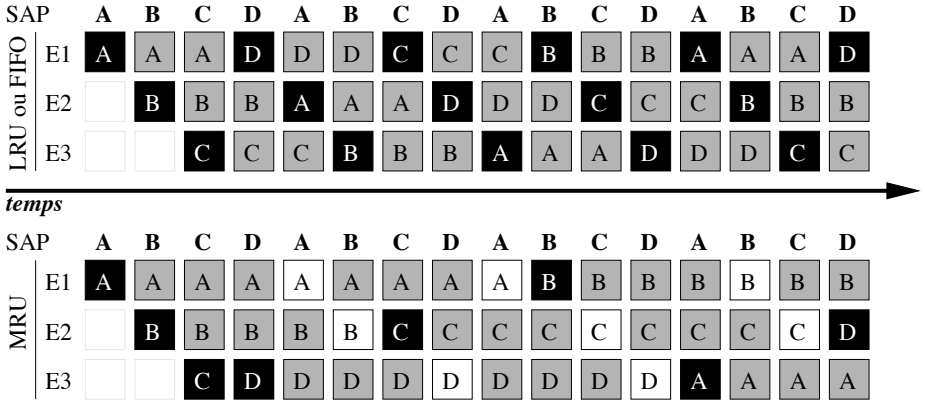
La politique MIN est proposée dans le contexte idéal où l'identité et l'ordre des pages qui vont être accédées est connu à l'avance [Bel66]. Elle repose sur la notion de *distance d'accès*. La distance d'accès à une page P au temps t_0, est le nombre de pages accédées entre le temps t_0 et le moment où P est accédée. La politique MIN consiste à remplacer la page qui a la plus grande distance d'accès, c'est-à-dire celle qui sera à nouveau accédée le plus tardivement. La figure 3.6.(b) illustre son comportement pour une séquence d'accès aux pages (SAP) quelconque.

Cette politique a la propriété de générer, pour n'importe qu'elle SAP, le minimum de défauts de page. Elle sert ainsi de borne théorique aux politiques de remplacement qui n'ont pas les moyens de connaître à l'avance et exactement la SAP des traitements.

Les politiques de remplacement suivent généralement une approche probabiliste. Elles supposent que la probabilité d'accès aux pages résidantes n'est souvent pas



(a) comportement de la politique MIN sur un parcours quelconque



(b) Comportement des politique LRU, FIFO et MRU pour un parcours séquentiel cyclique

FIG. 3.6 – Exemples de comportement de politiques de remplacement.

répartie de manière uniforme[1]. Plutôt que de prédire la SAP des traitements, elles essaient d'estimer la distribution de la probabilité d'accès, espérant que la page qui a le moins de chance d'être accédée soit celle ayant la plus grande distance d'accès.

La politique de remplacement doit être judicieusement choisie en fonction du type d'accès. Comme nous allons le voir, une politique, efficace pour certaines SAP peut être néfaste pour d'autres. La politique RAND, qui remplace les pages au hasard, sert de référence quant à l'efficacité d'une politique de remplacement. Une politique est efficace si elle parvient à utiliser efficacement l'information dont elle dispose de manière à engendrer moins de défauts de page que la politique RAND (qui elle ne nécessite aucune information). Notons que la politique RAND est particulièrement

[1]Elle suppose également que certaines pages résidantes ont une probabilité non nulle d'être à nouveau accédées. Dans le cas contraire où chaque page n'est accédée qu'une fois, le nombre de défauts de page est de toute façon indépendant de la politique choisie.

adaptée au cas où les accès sont uniformément répartis.

Le principe de localité temporelle stipule que plus une page a été récemment accédée plus elle a de chance de l'être à nouveau. Ce type de comportement est notamment vérifié lorsque les traitements accèdent des données qui ont été judicieusement regroupées (*clustering*). La politique LRU (Least Recently Used) [Spi76] qui remplace la page la moins récemment référencée convient à ce type de comportement. Cette politique possède en outre la propriété (dite *de Belady*) d'être conservatrice dans la mesure où, pour n'importe quelle SAP, son efficacité est une fonction croissante de la taille de la mémoire [BNS69] (ce qui n'est pas forcément le cas des autres politiques). De nombreuses variantes (comme CLOCK ou TWO-HAND-CLOCK), ont été utilisées pour la gestion mémoire des systèmes d'exploitation.

De nombreux travaux montrent cependant son inadéquation au contexte des bases de données [Kap80, Sto81, EH84]. Une des raisons est que les jointures par boucles imbriquées sont des opérations courantes dans ce contexte-là. Ce type d'opération a la particularité de parcourir de manière séquentielle et cyclique une des deux relations jointes. Comme le montre la figure 3.6, la distance d'accès d'une page est d'autant plus grande que cette dernière a été accédée récemment. La politique LRU remplace alors systématiquement la prochaine page accédée !

Pour ce type de parcours, la politique MRU (Most Recently Used), qui remplace la page la plus récemment accédée, est optimale puisqu'elle remplace la page qui a la plus grande distance d'accès (cf. figure 3.6.(b)). Elle n'est cependant adaptée que pour ce type de comportement.

La politique FIFO (First In First Out) suppose que plus la date de chargement d'une page est éloignée, moins elle a de chance d'être accédée dans un futur proche. Cette politique, très facile à implanter, convient particulièrement aux parcours séquentiels ponctués de légers retours en arrière. Elle convient également comme politique de remplacement des caches de journaux[2] du fait qu'elle remplace en priorité les plus vieilles pages et limite ainsi les pertes en cas de pannes [OV99]. Notons que, comme le montre la figure 3.6.(b), cette politique a le même comportement que la politique LRU pour des parcours séquentiels cycliques.

Une autre façon d'estimer la probabilité d'accès à une page est de compter le nombre de fois où elle a été accédée depuis qu'elle a été chargée en mémoire. C'est ce que fait la politique LFU qui remplace la page la moins référencée. Cette politique est

[2]Un journal est une suite d'enregistrements contenant des informations utiles à la mise en oeuvre d'un mécanisme de tolérance aux pannes. Afin de minimiser le coût d'écriture et la place du journal sur disque, les enregistrements peuvent être conservés en mémoire avant d'être écrits sur disque. Le serveur doit cependant assurer que tous les enregistrements concernant une transaction doivent être écrits avant validation de cette dernière (principe du Commit Rule).

particulièrement adaptée aux situations où la probabilité d'accès à une page est indépendante de l'identité des pages qui viennent d'être accédées. Prenons par exemple l'utilisation d'un index pour retrouver les n-uplets d'une relation. La structure d'un index est souvent un B-arbre, arbre équilibré pour lequel chaque noeud occupe une page logique et à un noeud d'un niveau correspond q noeuds du niveau inférieur (le nombre de noeuds de chaque niveau croît donc de manière exponentielle). La recherche d'un n-uplet nécessite le parcours de l'arbre en profondeur impliquant l'accès à un seul noeud de chaque niveau. La probabilité d'accéder à nouveau à un noeud dépend du niveau de l'arbre auquel il appartient. Si p est la profondeur de l'arbre, la probabilité d'accéder un noeud de niveau i est $\frac{1}{pq^i}$. Autrement dit, un noeud d'un niveau i a q fois plus de chance d'être à nouveau accédé qu'un noeud de niveau $i+1$, indépendemment de l'identité des noeuds qui viennent d'être accédés.

La politique LRU conserve grossièrement autant de noeuds de chaque niveau alors qu'il convient mieux de conserver en priorité les noeuds des niveaux supérieurs. La politique LFU y parvient cependant puisqu'elle conserve en priorité les pages qui ont été les plus référencées. Celle-ci se montre également efficace lorsque la distribution des accès suit la loi de Zipf [Zip49], ce qui est généralement constaté dans le contexte des serveurs et proxy Web [NHM+98, RV98, BCF+99]. Le reproche principal qui lui est fait est de ne pas être adaptée aux variations de comportement, notamment de conserver trop longtemps des pages qui ont été beaucoup accédées pendant une courte période mais ne l'ont plus été depuis longtemps. Des variantes se rapprochant de la politique LRU ont été proposées afin de limiter cette lacune [RD89, NDD92, LCK+99]. Bien que ces politiques réagissent mieux aux changements de comportement, elles restent cependant moins adaptées que la politique LRU pour des parcours à forte localité temporelle et les politiques FIFO et MRU pour des parcours séquentiels.

La politique LRU-K [OOW93, OOW99] est également un bon exemple de politique qui essaie de regrouper les avantages des politiques LRU et LFU. Elle consiste à remplacer la page dont le k dernier accès est le plus ancien. Pour k égal à 1, cette politique est équivalente à la politique LRU. En revanche, plus k est grand, plus elle tend à préserver les pages les plus fréquemment référencées telle que le fait la politique LFU. Les auteurs remarquent le bon comportement général de cette politique lorsque k est égal à 2 et plus particulièrement pour les parcours indexés[3]. Il montrent en outre que sa vitesse de réaction aux changements de comportement baisse à mesure que k augmente. Son efficacité est également limitée dans des situations particulières telles que les parcours séquentiels cycliques pour lesquels la politique MRU reste préférable.

[3]Une variante intéressante de LRU-2, 2Q, est présentée dans [JS94].

	LRU	MRU	FIFO	LRU-K	LFU
séquentiel cyclique	– –	++	– –	–	– –
séquentiel avec retour	+	–	++	+	+
principe de localité	++	– –	–	+	–
parcours d'index	– –	– –	– –	++	++

TAB. 3.1 – Efficacité des politiques LRU, MRU, FIFO et LRU-K selon le type de parcours.

Le tableau 3.1 résume, pour différents type de SAP, le comportement des politiques de remplacement que nous venons de présenter. Deux '+' indique un SAP de prédilection pour laquelle la politique est particulièrement adaptée. Un '+' indique que la politique réagit bien mais qu'il existe une politique plus adaptée. Un '-' indique un mauvais comportement tandis que deux '-' indique un comportement désastreux.

Nous avons présenté ici les politiques de remplacement les plus courantes. Bien d'autres ont été proposées dans la littérature pour des contextes particuliers. Nous citerons par exemple la politique L/MRP dans le contexte multimédia [MKK95], la politique BROOM basée sur des techniques de data-mining [TTL98], les politiques SEQ [GC97, SKW99] et EELRU à mi-chemin entre LRU et MRU, des politiques adaptées aux contraintes temps-réel [HS90], etc.

3.3.2 Remplacement en mémoire répartie

Le problème du remplacement prend une dimension supplémentaire lorsque nous nous plaçons dans le contexte d'une mémoire répartie. L'utilité d'une copie d'une page P dépend en effet du placement de la copie, de l'existence d'autres copies et de la distribution de la probabilité d'accès à P sur les différentes machines. L'approche la plus courante consiste à limiter le nombre de copies en mémoire afin de maximiser le pourcentage de la base en mémoire dans l'objectif de limiter le nombre d'E/S. Les pages répliquées sont généralement remplacées en premier puisque leur GAR est inférieur à celui des pages non répliquées.

La politique proposée dans [FCL92] se place dans un contexte où une machine serveur centralise l'accès à la base. Une machine cliente exécutant des traitements et désirant récupérer une copie d'une page passe systématiquement par le serveur[4]. Chaque machine possède sa propre politique de remplacement, à savoir LRU pour une

[4]Cette architecture asymétrique de type client/serveur centralisé n'est pas scalable mais la politique proposée ainsi que les suivantes sont intéressantes dans la mesure où elles tirent profit de la mémoire répartie afin de limiter le nombre d'E/S.

machine cliente et LIFO[5] pour la machine serveur. Lorsqu'une demande de copie de la page P ne peut être satisfaite par la mémoire du serveur, ce dernier demande à un client possédant une copie de P de satisfaire la demande à sa place. Cette méthode permet ainsi de profiter du partage (entre un client et le serveur et entre deux clients) afin de réduire le coût des défauts de page. La politique tente de limiter la réplication entre le serveur et ses clients mais pas entre les différents clients.

Deux extensions de cette politique sont proposées dans [DWAP94]. La première extension consiste à réserver statiquement une partie de la mémoire des machines clientes pour le compte du serveur. Cette approche tend à limiter le risque de réplication entre les clients. Cependant, un client peut être amené à contacter le serveur pour récupérer des copies qui se trouvent être dans sa propre mémoire locale mais pour le compte du serveur ! Dans une deuxième approche, appelée N-CHANCE, la politique de remplacement des clients conserve en priorité les copies uniques. Lorsqu'une machine ne possède plus que des copies uniques, la moins récemment utilisée est transférée au hasard vers une autre machine. Si la page est transférée plus de N fois sans être accédée, elle est détruite au même titre qu'une copie répliquée. De cette manière, la politique tend à minimiser encore plus le risque de réplication.

Les politiques proposées dans [VLN95] considère une architecture client/serveur symétrique où chaque machine se comporte à la fois comme le serveur d'une partie de la base et comme un client pour l'ensemble des données. Dans cette configuration, la mémoire d'une machine sert à conserver des copies de pages stockées localement (PSL) et de pages stockées à distance (PSD). Lorsque remplacée, une PSD et renvoyée à son serveur. Elle a donc une chance supplémentaire d'être conservée en mémoire répartie alors qu'une PSL devra être lue sur disque au prochain accès. Si la politique LRU est appliquée à l'ensemble des pages et que l'activité des traitements locaux est élevée, les PSL finissent par disparaître et le nombre global d'E/S augmente. Pour cette raison, la politique proposée dans [VLN95] garantie un minimum de place mémoire aux PSL même si leur probabilité d'accès est inférieur à certaines PSD. Une extension est proposée dans [VNL98] afin d'augmenter les chances qu'une PSD remplacée soit effectivement retenue par le serveur qui la gère. Pour cela, la politique remplace en priorité une PSD dont le serveur est peu chargé.

Hormis la politique N-CHANCE, les politiques de remplacement précédentes ne transfèrent une copie que vers une machine bien définie, en l'occurrence le serveur qui stocke la page. Si une machine est peu active tant d'un point de vue client que serveur,

[5]Comme nous allons le voir, la politique LIFO (Last In, First Out), inverse de la politique FIFO se justifie par le fait que la dernière page envoyée à un client est celle qui a le plus de chance d'être encore répliquée.

la mémoire dont elle dispose est sous-utilisée. La politique proposée dans [FMP+95] suit une approche similaire à N-CHANCE à la différence que la machine vers laquelle est transférée une copie est choisie en fonction d'une estimation de l'âge global des pages. Chaque machine possède ainsi à la fois des copies de pages utiles localement (PUL) et de pages utiles à distance (PUD). Les PUL peuvent être répliquées tandis que les PUD ne le sont pas. Lorsque l'activité d'une machine augmente, les PUD sont progressivement remplacée par des PUL et inversement, lorsqu'elle diminue, les PUL sont progressivement remplacées par des PUD. L'activité d'une machine se mesure par le proportion de PUL parmi celles qui ont été les plus récemment accédées de manière globale. En d'autres termes, la politique tente d'approcher une politique LRU globale tout en autorisant la réplication et sans tenir compte du GAR des copies.

L'approche présentée dans [SW97] s'appuie sur un modèle de coût afin de mieux estimer l'utilité des pages en fonction de leurs probabilités d'accès et de leurs GAR respectifs. Le modèle tient notamment compte de l'influence de la charge de traitement des machines sur le temps d'accès à la mémoire distante. Ce modèle est utilisé par la politique de remplacement dans l'objectif de minimiser le temps de réponse moyen des traitements.

Dans [LWY93], le problème de la gestion des copies est abordé de manière à prendre en compte différents objectifs de performance. La politique de remplacement tente de limiter le nombre et le coût des défauts de pages en fonction de l'objectif visé. Lorsque l'objectif est de maximiser le débit des traitements, maximiser l'utilisation du processeur et minimiser le nombre d'E/S est suffisant. Lorsqu'en revanche la manière dont sont ordonnés les traitements a une importance, la politique de remplacement doit tenir compte dans ses décisions des ordonnancements qu'elle rend possibles. Cela est par exemple important lorsque l'objectif de performance visé est de minimiser voire garantir un temps de réponse, assurer des débits ou temps de réponse équitables entre les différentes machines ou doit assurer la non perturbation de traitements locaux par le comportement de traitements distants (autonomie). Ils proposent un modèle de coût prenant en compte la probabilité d'accès aux copies ainsi que leur GAR et sur lequel s'appuient les décisions du GMR.

3.4 Politiques de préchargement

Différentes propositions de politique de préchargement sont présentées dans cette section. Toujours dans un souci de clarté, nous présentons d'abord le problème en mémoire locale puis en mémoire répartie. Une politique de préchargement en mémoire

locale doit décider *quelles* pages doivent être chargées à l'avance et *quand* elle doivent être chargées alors qu'en mémoire répartie elle doit également décider *d'où* et *vers où* précharger les données.

3.4.1 Préchargement en mémoire locale

Pour décider quelles pages précharger, une politique de préchargement s'appuie sur une technique de prédiction progressiste. Cette technique a pour but de prédire des séquences d'accès aux pages, c'est-à-dire quelles pages vont être accédées, quand elles vont être accédées et avec quelle probabilité. Les différentes approches varient fortement selon le type d'information qu'elles utilisent et analysent et le moment où la SAP est prédite. La prédiction des accès peut être implantée sous la forme d'une stratégie, déduite par apprentissage, par analyses de traces ou de code ou encore avisée par les programmes.

La politique de préchargement la plus utilisée et la plus simple à mettre en oeuvre consiste à simplement précharger les prochaines pages logiques de celles qui viennent d'être accédées (en suivant l'ordre d'adressage de l'espace logique). La politique OBL (One Block Lookhead) [Smi78] suit cette approche. Elle suppose qu'un programme qui accède de manière séquentielle un ensemble de pages logiques a de fortes chances de poursuivre avec ce même comportement. Un traitement qui effectue une recherche sans s'appuyer sur des méthodes d'accès structurées ou applique un ensemble d'opérations de manière uniforme à un ensemble de données a typiquement ce type de comportement. De ce fait, les systèmes d'exploitation emploient généralement cette politique dans la gestion de la mémoire virtuelle dès lors qu'ils détectent un accès séquentiel. La politique est en revanche néfaste pour des traitements parcourant un ensemble de données dans un ordre différent de celui de l'espace logique.

Les politiques reposant sur de l'analyse de traces partent du principe que les traitements accèdent souvent les mêmes données de la même manière. Elles supposent que les mêmes causes produisent les mêmes effets, c'est-à-dire, d'un point de vue SAP, que l'accès à un ensemble de pages entraîne l'accès à un autre ensemble. En connaissant à l'avance le comportement d'un traitement et en déterminant son état d'exécution, il est donc possible de prédire quelles pages il risque d'accéder et avec quelle probabilité. La prédiction des accès est généralement effectuée à l'avance indépendemment de l'exécution du serveur. Elle s'appuie sur des collections d'informations statistiques (traces) concernant le comportement passé des traitements se présentant généralement sous la forme d'une ou plusieurs longues séquences d'accès. Le problème à résoudre est d'extraire de ces gros volumes de données des informations pertinentes

et exploitables ensuite par le GMR. Certaines approches consistent à construire des graphes de précédance où les noeuds sont des pages et les arcs sont pondérés par la probabilité d'accéder une page après une autre [TD91, GA94, Bar98]. D'autres approchent consistent à utiliser des techniques de data-mining pour découvrir des patrons d'accès [PZ91, TTL98].

Dans certains contextes, le comportement des traitements est connu à l'avance, et il est possible d'aviser le GMR de certains types d'accès prévisibles. C'est par exemple le cas des programmes de calcul et de visualisation scientifique, les systèmes de reconnaissance de la parole ou des applications multimédia [PGG+95]. Si cette approche nécessite l'implication du programmeur et la communication entre les programmes et le GMR, les bénéfices peuvent être en revanche très spectaculaires du fait de la précision et l'exactitude des prédictions.

Une autre approche consiste à laisser cette charge à un compilateur spécifique capable de détecter des séquences d'accès particulières et de les communiquer au GMR de manière totalement transparente au programmeur [MDK96]. Cette approche a également été étudiée dans le domaine des bases de données pour lesquels les chemins d'accès peuvent être connus à l'avance au moment de l'optimisation et/ou l'exécution des requêtes [WZ86]. Il en est de même des systèmes de présentation multimédia pour lesquels les parcours de vidéo et les schémas de présentation permettent de prédire l'ordre dans lequel sont accédées les données [DS94, RMSW97, MKK95]. Le préchargement est dans ce cas d'un apport essentiel pour garantir les contraintes de continuité de flots.

Dans la mise en oeuvre de la politique de préchargement, le GMR est malheureusement confronté à deux propriétés antagonistes. Plus il anticipe le chargement à l'avance plus il parvient à réduire le coût des défauts de page (jusqu'à un certain point toutefois). Malheureusement, la prédiction de l'accès à une page est généralement d'autant plus fausse qu'elle se projette dans le futur. Cela peut se concrétiser par une mauvaise estimation de la probabilité mais également du moment de l'accès. Charger inutilement une page ou la charger trop tôt peut entraîner dans les deux cas le remplacement de pages résidantes jugées utiles et la surcharge du gestionnaire d'E/S.

Une étude menée dans [CFKL95] montre que même avec une connaissance exacte du futur, initier trop tôt le chargement d'une page peut avoir des conséquences plus néfastes que bénéfiques. L'exemple sur lequel s'appuie cette étude est illustré par la figure 3.7 et suppose une politique de remplacement optimale. Initialement la mémoire (constituée de deux emplacements) contient une copie des pages A et B. Lorsqu'aucun préchargement n'est effectué, le défaut de la page C entraîne le remplacement de la

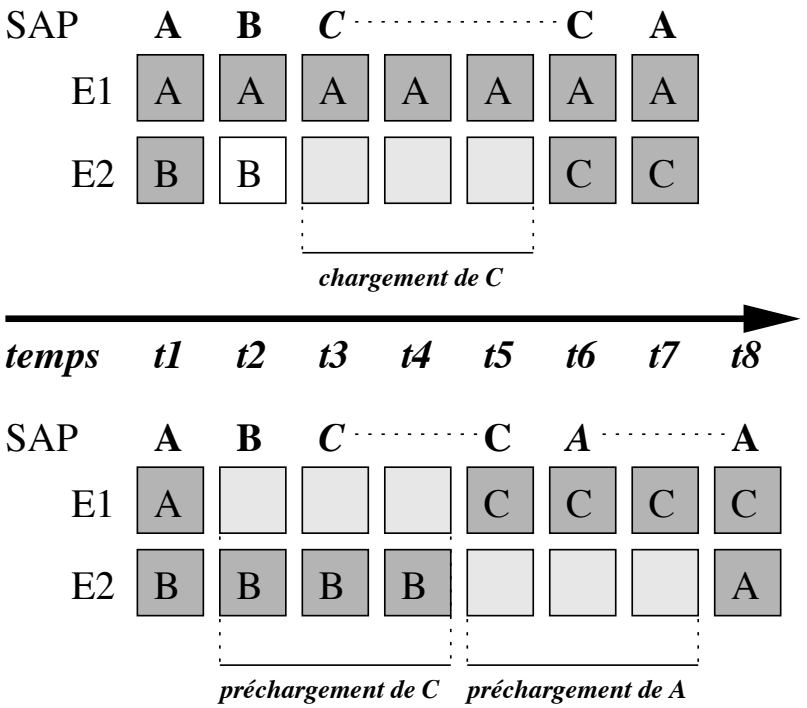


FIG. 3.7 – Augmentation du nombre de défauts de page dû à un préchargement trop précoce.

page B qui est celle qui a la plus grande distance d'accès. En anticipant le chargement de C d'une unité de temps, la page remplacée n'est plus B mais A. De ce fait, A devra être chargée à nouveau de sorte que même si ce chargement peut lui-aussi être anticipé, le temps total d'exécution est supérieur à celui obtenu sans préchargement. La conclusion des auteurs est qu'il est préférable d'intégrer les politiques de préchargement et remplacement. Elle argumente donc encore plus pour que la gestion du préchargement soit effectuée par le GMR.

3.4.2 Préchargement en mémoire répartie

Bien que cette approche n'a été que peu étudiée, le préchargement en mémoire répartie offre des possibilités supplémentaires appréciables. Comme le montre la figure 3.8, la politique de préchargement peut décider d'où et vers où précharger les données :

- (a) de l'espace de stockage vers la mémoire locale
- (b) de l'espace de stockage vers une mémoire distante
- (c) d'une mémoire distante vers la mémoire locale

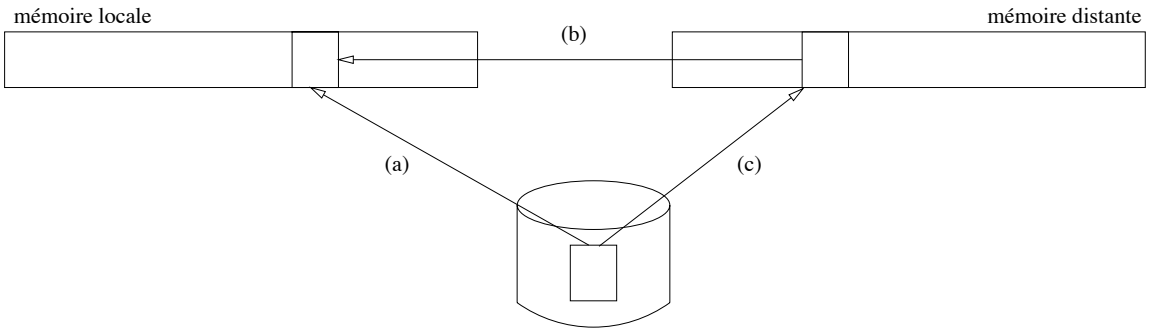


FIG. 3.8 – Préchargement en mémoire répartie.

Précharger une page vers la mémoire distante plutôt que directement vers la mémoire locale consiste à tirer profit de la disparité de charge sur l'ensemble des machines et permet de limiter l'impact d'un mauvais préchargement. En effet, la page préchargée remplace alors une page distante dont l'utilité est a priori moindre qu'une page locale. Le préchargement de la mémoire distante vers la mémoire locale est intéressant puisqu'il tire profit du partage de données. De plus, une étude menée dans [Bar98] montre que pour que le préchargement d'une page soit bénéfique, la prédiction doit être exacte à 98 % si les données sont chargées du disque et seulement 25 % si les données proviennent d'une mémoire distante.

Notons que ces différentes possibilités de préchargement peuvent permettre de mieux balancer les charges processeur, réseau et disque occasionnées par le chargement et le transfert de copies. Par exemple, si le réseau et/ou la machine sur laquelle se trouve une copie de la page est saturé, il peut être plus intéressant de précharger la page depuis le disque où est elle stockée directement vers la mémoire locale. Inversement, si le disque est saturé, il peut être plus intéressant de précharger la page depuis la mémoire distante. Enfin, si le processeur local est temporairement saturé de traitements, il peut être plus intéressant de précharger la page vers une machine distante, notamment s'il s'agit d'un disque partagé.

3.5 Conclusion

Nous avons présenté dans ce chapitre les principes de gestion de copies. L'objectif du GMR est de faire en sorte que les blocages des traitements occasionnés par les défauts de page contraignent le moins possible les possibilités d'ordonnancement permettant ainsi d'approcher le plus possible l'objectif de performance visé. Les méthodes utilisées consistent à tirer profit de la mémoire répartie et de la prédiction des accès, et d'anticiper le chargement des pages. Il met pour cela en oeuvre une politique

de remplacement et de préchargement pour lesquelles nous avons présentées quelques propositions.

De cette étude, nous pouvons constater que chaque politique de remplacement est particulièrement adaptée à un type d'accès particulier mais l'est moins pour d'autres types d'accès. Concernant les politiques de préchargement, nous pouvons remarquer que les méthodes de prédiction employées dépendent du type et de la quantité d'information disponible concernant le comportement des traitements.

Nous avons également montré à travers ces deux type de politiques l'intérêt d'une gestion répartie de la mémoire dans le but de mieux répartir la charge des différentes ressources matérielles. Cette capacité est d'autant plus nécessaire lorsqu'implanter un serveur scalable est un critère important.

Chapitre 4

Gestion du partage en mémoire répartie

4.1 Introduction

4.1.1 Contexte d'étude

Dans le chapitre précédent, nous avons considéré l'ensemble des traitements comme un seul traitement réparti. Dans ce chapitre nous faisons au contraire l'hypothèse que les traitements n'ont pas les mêmes besoins en terme de performance et/ou de qualité de service (QoS), n'ont pas forcément les mêmes comportements et doivent être vus comme des traitements indépendants. Nous disons que ces traitements sont *concurrents* dans le sens où le comportement et les besoins de chacun peuvent avoir une influence (pouvant être néfaste) sur les autres. Nous nous intéressons alors aux problèmes de *partage* de l'espace physique et de l'espace logique qui doivent être résolus à la fois pour des raisons de performances que de QoS.

Nous relâchons également l'hypothèse selon laquelle les différentes copies d'une page répliquée ont toutes la même valeur. Nous disons que deux copies d'une même page qui n'ont pas la même valeur *divergent*. En mémoire répartie comme en mémoire centralisée, la réplication peut être un moyen d'aller vers de meilleures performances. La divergence des copies doit cependant être prise en compte afin de garantir certaines qualités de service.

4.1.2 Motivation

La différenciation des traitements se justifie par le fait qu'ils ne sont pas forcément effectué dans un même but, ne participent pas forcément à la réalisation d'une tâche

commune et qu'ils n'ont pas forcément le même comportement ni les mêmes besoins.

Prenons l'exemple du serveur d'un système de gestion de base de données (SGBD) qui exécute simultanément un ensemble de requêtes pour le compte de différents utilisateurs. Deux requêtes n'ont pas les mêmes besoins en terme de ressources mémoires si l'une effectue la jointure de plusieurs grandes relations non indexées et l'autre effectue une simple sélection/modification portant sur la clé d'une petite relation indexée. Elle n'ont pas non plus le même comportement si la première parcourt de manière cyclique un ensemble de relations pendant que l'autre parcourt un B-arbre.

À ces différences peut s'ajouter le fait que les deux requêtes peuvent être impliquées dans des programmes différents. La première requête peut être exécutée dans le but d'extraire des informations statistiques de la base et son temps de réponse n'est pas un problème en soi (on attend le temps qu'il faut). L'autre requête, en revanche, peut faire partie d'une transaction ayant des contraintes de type temps réel et nécessiter un temps de réponse très court. De même, la "fraîcheur" et la cohérence des données peut ne pas être une nécessité pour la première requête et être d'une importance critique pour la deuxième.

4.1.3 Rôle du GMR

Les cas de figure évoqués ci-dessus montrent qu'il est important de pouvoir isoler les traitements les uns des autres. Le GMR doit assurer ce rôle en contrôlant le partage de l'espace physique et de l'espace logique. La gestion de la concurrence d'accès à ces deux espaces doit être effectuée de manière à faire cohabiter les différents objectifs de performance et les contraintes de qualité de service sur la même plateforme d'exécution.

Le partage de l'espace physique est traité dans la section 4.2. Le rôle du GMR est de mettre en oeuvre une *politique d'allocation* prenant en compte les besoins de chacun des traitements de manière à leur assurer un certain niveau de "confort d'exécution" et d'isoler leurs comportements respectifs. Nous verrons de plus que l'allocation explicite permet d'informer les traitements afin qu'ils puissent s'adapter aux ressources dont ils disposent.

Le partage de l'espace logique pose des problèmes de *cohérence et de synchronisation* (C&S) d'accès aux données que nous abordons dans la section 4.3. La synchronisation est nécessaire lorsque certaines qualités de services telles que l'isolation des transaction doit être assurée et qu'il existe un risque non nul de partage des données. À cela s'ajoute des problèmes de cohérence lorsque les pages sont répliquées. Le GMR doit s'assurer que la divergence des copies n'est là encore pas contradictoire avec la

qualité de service prétendue.

4.2 Politiques d'allocation

4.2.1 Principes

L'allocation de la mémoire physique consiste à décider comment distribuer l'ensemble des emplacements aux différents traitements qui s'exécutent. L'*espace d'allocation* d'un traitement est constitué de l'ensemble des copies créées ou conservées pour le compte du traitement. Une copie peut changer d'espace d'allocation au cours du temps. Par exemple, une copie créée pour le compte d'un premier traitement peut être conservée pour le compte d'une deuxième traitement après la terminaison du premier. La copie aura alors appartenu d'abord à l'espace d'allocation du premier traitement puis à celui du second.

Une copie peut être utile simultanément à plusieurs traitements à la fois. Toutefois, les métriques prises pour mesurer l'utilité pour le compte de chaque traitement peuvent ne pas être comparables. Nous appelons *traitement principal* d'une copie le traitement dont l'espace d'allocation contient la copie et *utilité principale* d'une copie son utilité pour ce traitement. Lorsque le GMR juge que l'utilité principale d'une copie ne justifie pas de la conserver, il peut toutefois juger de son utilité pour les autres traitements. La copie peut alors être conservée en mémoire en la changeant d'espace d'allocation.

4.2.2 Avantages d'une gestion explicite de l'allocation

L'allocation explicite consiste à contrôler complètement les espaces d'allocation associés aux traitements plutôt que de laisser ce soin à une politique de remplacement globale. Conceptuellement, la gestion du partage de l'espace physique peut être vue comme un partitionnement de celui-ci où chaque partition représente un espace d'allocation. Ce partitionnement permet d'isoler les comportements des traitements les uns des autres et de mieux les informer sur la quantité de mémoire qui leur est allouée.

4.2.2.1 Isolation des comportements

L'entrelacement des traitements rend difficile une prédiction globale des accès. Plus les traitements sont entrelacés, plus la "somme" de leurs comportements résulte en un comportement chaotique peu exploitable. Il est dans ce cas difficile d'estimer

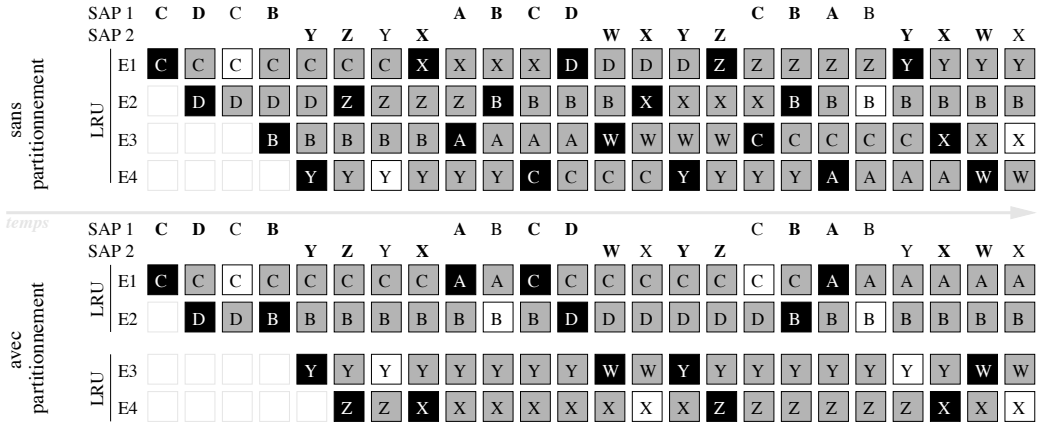
l'utilité d'une copie et par conséquent d'appliquer une politique de remplacement globale. Remarquons en outre que dans un contexte d'exécution parallèle, un traitement est doublement pénalisé par un défaut de page. En plus du fait qu'il est bloqué pendant le temps de chargement de la page, l'utilité des copies appartenant à son espace d'allocation a de fortes chances de diminuer puisqu'elle ne peuvent pas être accédées. Cette diminution peut conduire ces pages à être remplacées en premier, augmentant les chances que le traitement essuie un nouveau défaut de page à sa ré-activation.

Cette section montre à travers différents exemples l'intérêt de partitionner la mémoire afin d'isoler mutuellement les traitements. Nous comparons pour chaque exemple l'application de la politique LRU sur l'ensemble des copies, et celle par partitionnement de la mémoire et utilisation de politiques de remplacement locales (LRU ou autres).

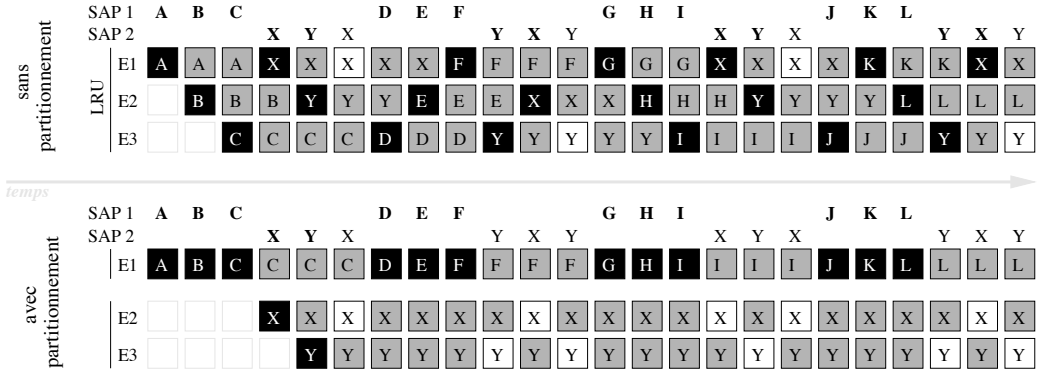
Comme le montre la figure 4.1.(a) partitionner la mémoire permet de préserver deux traitements exhibant chacun une localité de référence mais sur des données différentes. L'entrelacement des deux traitements résulte en un comportement pour lequel le principe de localité est moins significatif que pour chacun des traitements. Les pages chargées pour le compte d'un traitement sont peu à peu remplacées par celles chargées pour le compte de l'autre traitement. En attribuant explicitement deux emplacements à chacun et en appliquant la politique LRU sur chaque partition, le GMR parvient à réduire le nombre de défauts de page de 20 %.

Le partitionnement est d'autant plus avantageux lorsque les deux traitements n'ont pas le même comportement. L'exemple de la figure 4.1.(b) suppose qu'un des traitements respecte le principe de localité (SAP 2) tandis que l'autre effectue un parcours séquentiel (SAP 1). L'entrelacement des deux traitements ne résulte pas en un parcours séquentiel et n'exhibe pas non plus une forte localité de référence. Au détriment du deuxième traitement, la politique LRU conserve les pages du parcours séquentiel qui ne seront pas accédées à nouveau. Puisque de toute façon le parcours séquentiel entraîne systématiquement un défaut de page à chaque accès, il suffit de lui attribuer un emplacement et de laisser les autres pour le deuxième traitement. En procédant ainsi, le nombre de défauts de page est réduit de 30 %.

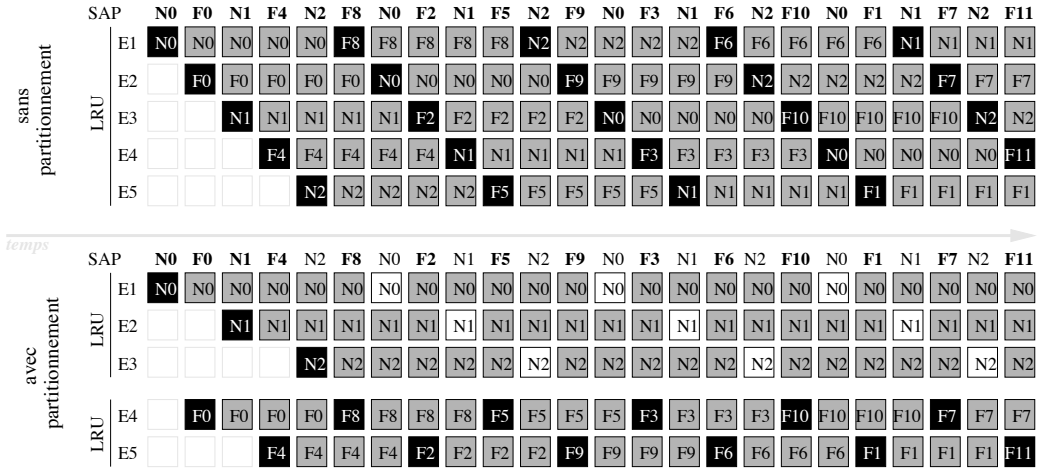
L'accès à une relation indexée peut être vu comme deux traitements coopératifs, l'un opérant sur l'index afin de retrouver une page de la relation et l'autre accédant la relation. Dans ce type de chemin d'accès, la probabilité d'accéder à nouveau à une page de l'index est plus importante que pour une page de la relation. Supposons une relation de 12 pages ($F0$ à $F11$) indexée à l'aide d'un B-arbre contenant un noeud racine et 3 noeuds au premier niveau ($N0$, $N1$ et $N2$). La probabilité d'accéder à nouveau à un noeud est ici quatre fois supérieur que celle d'accéder à nouveau à



(a) Préservation de la localité de référence



(b) Isolation de comportements antagonistes



(c) Isolation d'un index et de sa relation associée

FIG. 4.1 – Comportement de la mémoire avec et sans partitionnement.

une feuille. Malheureusement, comme le montre la figure 4.1.(c), les noeuds ne sont pas accédés suffisamment fréquemment pour qu'ils aient une chance d'être conservés par une politique LRU globale. En réservant dès le départ trois emplacements pour le compte des noeuds de l'index, le GMR parvient à les préserver de l'accès à la relation dont les pages ne sont ici accédées qu'une fois. Il en résulte une diminution de 37,5 % du nombre de défauts de page.

4.2.2.2 Adaptation des traitements

Lorsque le GMR gère explicitement l'allocation de l'espace physique, il est en mesure d'informer un traitement sur la taille de son espace d'allocation. Cette information peut ensuite être utilisée afin de choisir un algorithme adapté au nombre d'emplacements disponibles ou éventuellement d'adapter l'algorithme à mesure que la taille de l'espace d'allocation varie.

Un contexte permettant de bien comprendre comment les traitements peuvent s'adapter en fonction de la mémoire disponible est celui des bases de données. Avant d'être exécutée, une requête est compilée et optimisée. L'optimisation consiste à construire un plan d'exécution minimisant une fonction de coût. Le plan consiste en un arbre où les noeuds sont des opérateurs (parcours, sélection, jointure) et les feuilles sont les relations concernées par la requête. L'optimiseur ordonne les opérateurs de manière à minimiser la fonction de coût.

Généralement, le coût est une mesure du nombre d'entrées/sorties (E/S) qui devront être effectuées lors de l'exécution de la requête. Pour estimer le coût, l'optimiseur se base sur une estimation de différents paramètres : la taille des relations, la sélectivité des prédicats, la mémoire disponible, etc. Cette estimation peut cependant être très éloignée de la réalité. La taille des relations évolue dans le temps, les erreurs de sélectivité se propagent le long du plan d'exécution [IC91] et la mémoire disponible dépend de la charge imprévisible du serveur au moment de l'exécution de la requête. De nombreux travaux argumentent pour que des choix d'optimisation soient effectués juste avant ou pendant l'exécution des requêtes [CY89, CG94, SE93]. Par exemple, l'étude menée dans [CY89] montre que tenir compte de la mémoire disponible dans l'ordonnancement d'une série de jointures permet une réduction considérable du nombre d'E/S.

L'optimiseur choisit également les algorithmes implantant les opérateurs de manière à minimiser la fonction de coût. En fonction de l'estimation de la taille des relations R et S en entrée, de la mémoire disponible M et des méthodes d'accès sur les relations, l'optimiseur choisira un algorithme de jointure en une passe (nested-loop-

join, merge-join, etc.) ou en deux passes (hash-join, sort-merge-join, etc.) [GUW99].

Si $||S||+1$ emplacements sont disponibles[1], l'algorithme nested-loop-join réalise la jointure en effectuant $||R|| + ||S||$ E/S. Ce nombre croît cependant considérablement à mesure que le nombre d'emplacements diminue. Lorsque M est petit par rapport à $||S||$, le nombre d'E/S est de l'ordre de $||R|| * ||S||$ et un algorithme en deux passes est alors plus intéressant. Si $\sqrt{||S||}$ emplacements sont disponibles, l'algorithme hash-join réalise par exemple la jointure en générant un nombre d'E/S de l'ordre de $3 * (||R|| + ||S||)$. En connaissant la quantité de mémoire qui peut être alloué à la requête, il est possible de décider lequel de ces deux algorithmes est le plus intéressant.

De même, il arrive que deux opérateurs consécutifs puissent être exécutés en parallèle (*pipelining*). Cette approche permet de réduire le nombre d'E/S mais nécessite cependant plus d'espace physique. Une mauvaise estimation de la quantité de mémoire disponible peut conduire à un nombre d'E/S supérieur à celui d'une exécution séquentielle. Une alternative proposée dans [BKV98] est de décider du degré de parallélisme à l'exécution de la requête, c'est-à-dire lorsque le nombre d'emplacements effectivement disponibles est connu. Ceci montre encore un exemple où le GMR peut aider les traitements à s'adapter aux conditions réelles d'exécution.

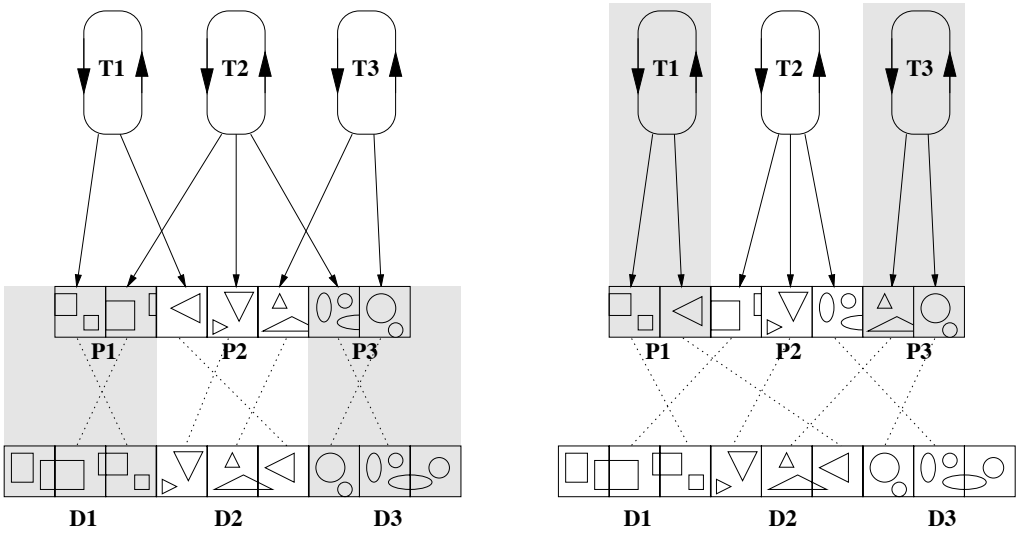
4.2.3 Choix de mise en oeuvre

4.2.3.1 Méthode d'association

Dans la section 4.2.2.1, nous avons vu l'intérêt du partitionnement pour isoler les comportements de différents traitements. Nous avons vu également qu'il permet d'isoler différents ensembles de données qui n'ont pas la même probabilité d'accès. Un domaine regroupe un ensemble de données ayant des caractéristiques d'accès communes [Rei76]. Ce concept permet par exemple de séparer les relations de leurs index associés, les données de la base des structures de gestion [EH84], différentes données en fonction de leur taille [MAM98], etc.

Le GMR peut donc décider d'allouer les emplacements en fonction du domaine des données accédées ou de l'identité des traitements. Ces deux possibilités sont illustrées par la figure 4.2. L'exemple montre trois traitements ($T1$, $T2$ et $T3$) accédant des données appartenant à trois domaines de données ($D1$, $D2$ et $D3$) dans un espace physique divisé en trois partitions ($P1$, $P2$ et $P3$). Le GMR peut choisir d'associer une partition Pi au traitement Ti ou de l'associer au domaine Di. Dans le premier cas, les pages accédées par Ti sont chargées dans la partition Pi indifféremment de

[1] $||R||$ désigne le nombre de pages logiques occupées par la relation R.



(a) Isolation des domaines

(b) Isolation des traitements

FIG. 4.2 – Modes d'isolation par partitionnement.

leurs domaines d'origine. Dans le deuxième cas, les pages provenant du domaine Di sont chargées dans Pi indifféremment des traitements qui les accèdent. Notons que ces deux approches peuvent être combinées. Par exemple, les politiques proposées dans [CD85, FNS95] créent pour chaque requête, une partition pour chaque relation parcourue.

Lorsque le GMR adopte une approche par isolation des traitements, il doit gérer les cas de partage d'une page logique par deux traitements. Les possibilités qui lui sont offertes sont de créer une copie différente dans chacune des partitions ou de n'en créer qu'une seule. La première approche peut conduire à une réplication inutile de la page notamment lorsque les deux traitements s'exécutent sur la même machine. La deuxième requiert que les deux traitements s'exécutent sur la même machine mais permet de mieux utiliser la mémoire physique. Le GMR doit alors décider comment traiter le problème du remplacement des copies physiquement partagées. Lorsque la politique de remplacement de la partition à laquelle appartient la copie décide de remplacer cette dernière, le GMR peut décider soit de détruire la copie, soit de lui laisser une seconde chance en la changeant de partition.

4.2.3.2 Méthode d'allocation

Une première façon de gérer le partitionnement de l'espace physique est d'imposer que les emplacements d'une partition appartiennent tous à la même mémoire locale.

Lorsque les partitions sont associées aux traitements, la mémoire locale est celle de la machine sur laquelle s'exécute le traitement. Si les partitions sont rattachées aux domaines, seuls les traitements s'exécutant sur la machine de la partition peuvent accéder au domaine.

La distribution des emplacements consiste alors simplement à décider de la taille de chaque partition. La taille d'une partition Pi peut être contrôlée par deux valeurs min_i et max_i. La valeur min_i représente le nombre d'emplacements que le GMR garantit à la partition et permet d'assurer un niveau de "confort d'exécution". La valeur max_i fixe une limite à ce nombre d'emplacements et permet de limiter l'impact du comportement d'un traitement sur les autres traitements. Le GMR doit assurer qu'à tout moment l'inégalité $\sum_i min_i \leq M$, où M est la quantité de mémoire totale, est vérifiée. Cette contrainte limite le nombre de partitions pouvant coexister et, lorsque les partitions sont associées aux traitements, limite le nombre de traitements pouvant s'exécuter en parallèle. Dans le deuxième cas, nous supposons que l'ordonnanceur met en oeuvre une *politique d'admission* dont les décisions sont fonction de l'objectif de performance visé et/ou des contraintes temps-réel imposées.

À sa création une partition Pi se voit allouer c_i emplacements et nous notons n_i le nombre d'emplacements courant de la partition. À tout moment le GMR doit assurer que les inégalités $\forall i,\ n_i \leq max_i$ et $\sum_i n_i \leq M$ sont vérifiées. Notons que vérifier l'inégalité $\forall i,\ n_i \geq min_i$ n'est pas une obligation. Cela signifie que le GMR garantit qu'à tout moment la taille d'une partition peut atteindre la valeur min_i même si elle peut être temporairement inférieure à cette valeur. En fonction des valeurs que peuvent prendre les valeurs c_i, différents méthodes d'allocation sont alors possibles :

Allocation statique ($c_i = min_i = max_i$) La politique alloue dès le départ le nombre d'emplacements garantis. Cette politique est "gloutonne" dans la mesure où le traitement met forcément un certain temps avant d'utiliser tous ses emplacements conduisant à une sous-exploitation de la mémoire. La politique de remplacement est déclenchée dès que le nombre de pages accédées est supérieur à min_i.

Allocation garantie ($c_i < min_i = max_i$) Le nombre d'emplacements alloués peut être inférieur à celui garanti et croit dès que le nombre de pages accédées est supérieur à c_i. Comme précédemment, la politique de remplacement est déclenchée dès que le nombre de pages accédées est supérieur à min_i. Les emplacements non alloués constituent une partition de taille variable pouvant être utilisée comme "cache de deuxième chance". Lorsqu'une page d'une partition est remplacée, la copie est transférée dans cette partition. Si elle est de nouveau accédée avant

d'avoir été remplacée, une E/S est évitée.

Allocation dynamique ($c_i \leq min_i < max_i$) Le nombre d'emplacements alloués peut être supérieur à celui garanti. Lorsque le nombre de pages accédées dépasse min_i, la politique d'allocation décide si oui ou non des emplacements supplémentaires peuvent être accordés au traitement. De même, elle peut réclamer les emplacements supplémentaires à tout moment. La politique de remplacement est activée lorsque le nombre d'emplacements alloués dépasse max_i, lorsque la politique d'allocation refuse d'allouer un emplacement supplémentaire ou lorsque cette dernière décide d'en réclamer.

L'allocation dynamique offre bien entendu la méthode d'allocation la plus flexible mais est cependant plus difficile à mettre en oeuvre. Le temps qu'un traitement ayant une valeur min_i élevée atteigne son "rythme de croisière", les emplacements libres peuvent être utilisés par les traitements actifs. Comme nous le verrons dans la suite, elle permet également de diminuer la valeur min_i et d'assouplir ainsi l'admission des traitements. L'allocation statique, à peine plus simple à mettre en oeuvre que l'allocation garantie, ne permet pas d'exploiter temporairement les emplacements réservés mais non encore alloués. Lorsque les traitements tardent à utiliser leur emplacements garantis, une partie des emplacements sont temporairement alloués inutilement.

Gérer l'allocation de manière répartie consiste à permettre la création de partitions réparties, c'est-à-dire constituées d'emplacements appartenant à des machines différentes. Une partition peut alors être créée pour un traitement non réparti désirant profiter de mémoire distante sous-utilisée (comme dans l'approche proposée dans [DWAP94] et présentée dans la section 3.3.2), pour un traitement réparti (tel qu'un serveur) désirant s'isoler d'autres traitements (éventuellement répartis), ou pour un domaine pouvant être accédé depuis différentes machines. La politique de remplacement d'une partition est alors une politique répartie qui doit tenir compte du gain d'accès relatif des différentes copies de la partition.

Une partition Pi est alors composée des *portions* $p_{i,j}$ des mémoires des machines m_j. Le problème du contrôle de l'allocation de l'espace physique revient à un problème de gestion d'allocation de mémoires locales entre leurs portions respectives. Pour chaque m_j possédant une mémoire locale de taille M_j, le GMR doit assurer que les inégalités $\sum_i min_{i,j} \leq M_j$, $\forall i$, $n_{i,j} \leq max_{i,j}$ et $\sum_i n_{i,j} \leq M_j$ sont vérifiées, où $n_{i,j}$, $min_{i,j}$ et $max_{i,j}$ représentent respectivement le nombre d'emplacements courant, garanti et maximal de la portion $p_{i,j}$. Tout comme en mémoire centralisée, une allocation statique, garantie ou dynamique est possible.

4.2.4 Distribution des emplacements

La politique d'allocation du GMR doit décider du nombre d'emplacements accordés à chaque partition. Elle décide donc pour chaque partition Pi, des valeurs c_i, min_i et max_i si la partition est centralisée et des valeurs $c_{i,j}$, $min_{i,j}$ et $max_{i,j}$ si la partition est répartie. À notre connaissance, cependant, aucune politique d'allocation répartie n'a été proposée. Par conséquent, la suite de cette section présente uniquement différentes propositions de politique d'allocation en mémoire locale. Elles reposent sur une estimation des besoins en place mémoire des différents traitements dont la précision est un facteur déterminant de l'efficacité de la gestion mémoire. Allouer trop peu d'emplacements pour un traitement conduit à un excès d'E/S pour ce traitement pouvant conduire à son effondrement (*thrashing*) tandis qu'en allouer trop conduit à l'inexploitation de ressources mémoire qui pourraient être mises à profit pour d'autres traitements.

Le modèle de coût HOTSET [SS82, SS86] sert de base à l'optimiseur lors de la compilation des requêtes simples. Il repose sur le principe que le nombre d'E/S[2] d'une requête en fonction de la place mémoire allouée et consiste en une série d'intervalles stables $[a_{2k} \ldots a_{2k+1}[$ séparés par des intervalles instables $[a_{2k+1} \ldots a_{2k+2}[$, $k \in I\!N$. Les valeurs a_{2k}, appelées "points chauds", sont particulièrement intéressantes puisqu'elles correspondent à des valeurs d'allocation à partir desquelles l'ajout d'emplacements n'apporte pas d'amélioration significative jusqu'au prochain point chaud. Ils en concluent que le nombre d'emplacements alloués à une requête doit correspondre à un point chaud. La politique d'allocation HOTSET met en oeuvre une allocation garantie avec $c_i = 0$, $min_i = max_i = a_{2k}$ et $a_{2k} \leq M < a_{2k+1}$, et utilise les emplacements libres comme cache de deuxième chance.

Dans [CD85], la politique DBMIN effectue un partitionnement plus fin en associant une partition à chaque parcours de relation. Cette fois, la taille de chaque partition dépend du type de parcours effectué et ne nécessite pas la connaissance des points chauds. De plus, chaque partition peut être gérée avec une politique de remplacement spécifique (parmi LRU, MRU et FIFO) en fonction du type du parcours qui lui est associé. Quelques exemples d'allocation sont présentées dans le tableau 4.1. Notons que le nombre d'emplacements alloués pour un parcours aléatoire dépend de la valeur de b_{yao}[3]. Si $b_{yao} < M$ alors b_{yao} emplacements sont alloués, sinon un seul emplacement est alloué. L'allocation d'emplacements est similaire à celle de la politique HOTSET (politique garantie), seule la manière de calculer les valeurs min_i et max_i

[2]Le nombre d'E/S est estimé en supposant que la politique LRU est systématiquement appliquée à chaque partition.

[3]b_{yao} est une estimation du nombre moyen de pages accédées par un parcours aléatoire [Yao77].

Type de parcours	séquentiel	séquentiel cyclique	aléatoire	séquentiel groupé
Allocation ($min_i = max_i =$)	1	$\|\|R\|\|$	1 ou b_{yao}	$max_{c \in groupes(R)} \|\|c\|\|$
Remplacement	-	MRU	RAND	FIFO

TAB. 4.1 – Exemples d'allocation et de politique de remplacement avec la politique DBMIN.

étant différente.

Dans [NFS91, FNS95], les auteurs notent que l'utilisation de la politique MRU pour les parcours séquentiels cycliques a cependant peu d'intérêt si le GMR garantit que le nombre d'emplacements alloués est égal au nombre de pages de la relation (aucune page ne sera jamais remplacée !. Ils remarquent également que b_{yao} est généralement trop grand de telle sorte que le GC n'alloue jamais plus d'un emplacement à un parcours aléatoire. De plus, les valeurs d'allocation minimale des politiques HOTSET et de DBMIN sont jugés trop élevées car conduisant à un degré de parallélisme trop faible. Les auteurs proposent la politique d'allocation MARGINAL GAINS autorisant une exécution sous-optimale des requêtes, c'est-à-dire avec un nombre d'emplacements inférieur à celui garanti par les politiques précédentes. L'idée est que la durée d'une exécution sous-optimale peut être finalement inférieur à la somme des temps d'admission et d'exécution optimale. Notons que les auteurs choisissent une méthode d'allocation statique. Ce choix se justifie par le fait qu'en admettant plus facilement les requêtes, les emplacements libres ne sont pas suffisamment nombreux pour justifier un cache de deuxième chance.

La politique d'allocation dynamique MINMAX est proposée dans [PCL94] dans le contexte des SGBD temps-réels. L'allocation consiste à attribuer les emplacements en fonction de la priorité des requêtes et consiste en deux phases. La première alloue min_i emplacements à chaque traitement en commençant par les plus prioritaires puis distribue le restant dans le même ordre sans toutefois dépasser la valeur max_i. La politique est invoquée à chaque fois qu'une requête entre ou sort du système, et à chaque changement de priorité. Ainsi, le nombre d'emplacements alloués à une requête peut osciller entre min_i et max_i au cours de son exécution. Des algorithmes capables de supporter des variation d'allocation doivent cependant être utilisés pour implanter les opérateurs. Fort heureusement, de tels algorithmes ont été proposés en nombre dans la littérature [ZG90, PCL93, DG94, ZL97].

Nous arrêtons ici la liste des politiques d'allocation, bien d'autres pouvant être trouvées dans la littérature [BCL93, CR93, MD93, YC93, DG95, BCL96]. Toutes montrent l'intérêt de contrôler explicitement l'allocation de la mémoire et d'appliquer des politiques de remplacement localement à chaque partition plutôt que de laisser

une politique de remplacement globale contrôler l'allocation des emplacements.

4.3 Politiques de cohérence et de synchronisation

4.3.1 Principes et objectifs

Les problèmes de cohérence se posent lorsque le système admet la réplication des données et que les différentes copies d'une page répliquée peuvent diverger à cause de leur modification en parallèle par différents traitements. Afin de garantir certaines propriétés telles que la convergence ou la non-divergence des copies, le système peut être amené à synchroniser les traitements. Synchroniser les traitements peut être cependant nécessaire même lorsque les données ne sont pas répliquées à partir du moment où permettre des modifications non contrôlées sur une ou plusieurs pages peut conduire à un état incorrect de la base.

Les problèmes de cohérence de copies ont été étudiés parallèlement pour les systèmes offrant une mémoire virtuelle répartie partagée (MVRP) et des systèmes de gestion de bases de données réparties (SGBDR). Ces deux types de systèmes ont comme caractéristique commune d'autoriser l'accès en parallèle à des données répliquées dans un espace physique réparti. Dans ce contexte, une page change de valeur à chaque écriture et une lecture doit retourner la *dernière* valeur écrite. La notion de "dernière valeur" pose cependant un problème du fait qu'il est difficile de définir un temps absolu et que les modifications portent sur des copies différentes. Ainsi, une MVRP ou un SGBDR doivent mettre en oeuvre des mécanismes et politiques de gestion afin de garantir un *modèle de cohérence*. Du point de vue des traitements, un modèle de cohérence peut être vu comme une qualité de service (QoS) que le système garantit.

Dans le contexte des MVRP, un modèle de cohérence de référence est la *cohérence séquentielle* définie dans [Lam79]. Ce modèle stipule que l'exécution parallèle des accès dans la mémoire répartie doit être équivalente à une exécution séquentielle sur une mémoire centralisée. Le modèle repose donc sur une notion d'*équivalence* par rapport à un *ordre total* des accès aux pages. Le rôle du système est dans ce cas d'ordonnancer les accès et mettre en cohérence les copies divergentes.

La théorie sur laquelle repose la gestion de la cohérence dans le contexte des bases de données est la *sérialisabilité* des transactions [BHG87]. Une transaction consiste en une suite de lectures et d'écritures dont le début et la fin sont explicitement déclarés par le programme. La terminaison peut correspondre soit à une annulation soit à une demande de validation qui cependant peut être rejetée par le système et traduite

en une annulation. Une différence essentielle entre la sérialisabilité et la cohérence séquentielle est que la première porte sur l'ordonnancement d'un ensemble d'accès. Une exécution en parallèle d'un ensemble de transactions est dite sérialisable si elle est équivalente à une exécution séquentielle des mêmes transactions[4]. Ici encore, cette théorie repose sur une notion d'équivalence par rapport à un ordre total et le rôle du GMR est toujours d'ordonnancer correctement les traitements et de mettre en cohérence les copies divergentes des pages répliquées.

Dans un cas comme dans l'autre, le système doit être informé des accès aux pages et disposer des moyens nécessaires pour suspendre l'exécution des traitements. Les intentions d'accès peuvent être soit explicitement signalées par les traitements soit détectées par le système de manière totalement transparante. Afin de ne pas induire un coût déraisonable pour le maintient d'un modèle de cohérence, le système peut cependant tirer profit d'informations concernant le comportement des traitements. Par exemple certain traitements utilisent des primitives de synchronisations afin de s'assurer de l'exclusion mutuelle à une section critique ou signaler la mise à disponibilité d'un résultat exploitable. De telles synchronisations sont nécessaires afin de limiter l'ordonnancement de traitements partageant des données, qu'elles soient répliquées ou non. Ainsi, dans le contexte des MVRP, d'autres modèles de cohérence ont été proposés comme la cohérence faible (weak-consistency), la cohérence à l'entrée (entry-consistency) ou la cohérence au relâchement (release-consistency). Les protocoles implantant ces modèles assurent la cohérence séquentielle des variables de synchronisation et supposent que les programmes sont bien construits vis-à-vis des hypothèses comportementales (ils doivent par exemple utiliser un sémaphore afin d'entrer et sortir proprement d'une section critique). Ces modèles sont dits à "cohérence faible" dans la mesure où aucune propriété n'est garantie si les traitements ne respectent pas les règles imposées. En assurant une qualité de service inférieure mais toutefois acceptable, ils parviennent ainsi à libérer des ressources (CPU, réseau) pour l'exécution effective des traitements et sont donc avantageux indépendamment de l'objectif de performance visé.

Le même raisonnement est applicable au contexte des bases de données. En effet, que les données soient répliquées ou non, il est nécessaire de contrôler les accès aux données afin de garantir la sérialisation des transactions. Une manière d'opérer peut être d'obliger les transactions à poser des verrous sur les données qu'elles ont l'intention d'accéder. En mémoire répartie, bien que de nouveaux problèmes de cohérence se posent, il n'est pas nécessaire d'jouter de nouvelles contreintes comportementales afin

[4]Notons que lorsque les transactions se résument à un seul accès (lecture ou écriture d'une page), cohérence séquentielle et sérialisabilité sont deux modèles équivalents.

de mettre en oeuvre des mécanismes assurant la convergence des copies. Par exemple, si l'on prend une MVRP comme support d'adressage d'un SGBDR, garantir la cohérence séquentielle de la MVRP serait superflu et induirait un coût d'exploitation déraisonnable. Pour qu'une MVRP puisse permettre d'implanter un SGBDR, il est donc nécessaire qu'aucun modèle de cohérence ne soit imposé.

La mise en oeuvre d'un modèle de cohérence doit tenir compte des besoins des traitements afin d'offrir les meilleures performances possibles. Cela se mesure en terme d'utilisation des ressources comme nous venons de voir mais également en terme de possibilités d'ordonnancement. Ici encore, le GMR doit assurer que les contraintes d'ordonnancement imposées par la politique de Cohérence et Synchronisation (C&S) permettent de se rapprocher le plus possible d'un ordonnancement idéal[5]. Lorsque l'objectif est de maximiser le débit ou minimiser le temps de réponse des transactions, minimiser les blocages en nombre et en durée est généralement une condition suffisante. Les décisions prises par le GMR ne doivent cependant pas aller à l'encontre d'autres qualités de services telles que par exemple les contraintes de type temps-réel. Comme nous le verrons cela nécessite l'implantation de politiques de C&S adaptées.

4.3.2 Sérialisation des traitements

Bien que les politiques de C&S propres au contexte des MVRP soient intéressantes et peuvent avoir des liens avec celles proposées dans le contexte des bases de données, notre étude des problèmes de cohérence et de synchronisation porte essentiellement sur la sérialisation des transactions. Ce choix se justifie par le fait que dans le contexte des serveurs de données, il est souvent nécessaire d'assurer l'intégrité d'*ensembles* de données non pas simplement l'intégrité de données isolées.

4.3.2.1 Principes

Selon sa définition première, un ordonnancement est sérialisable lorsqu'il est équivalent à un ordonnancement séquentiel. L'idée est que si chaque transaction fait passer la base d'un état intègre à un autre état intègre alors une exécution sérialisable de transactions n'a pas de conséquence sur l'intégrité de la base. L'équivalence porte à la fois sur les effets produits sur la base de données (écritures) et sur la vue qu'en ont les transactions (lectures).

Une manière de s'en assurer et de faire en sorte que l'ordonnancement soit équivalent à une exécution séquentielle. Cette approche est généralement assimilée à la

[5]Dans ce contexte, un ordonnancement idéal correspond à un ordonnancement maximisant l'objectif de performance et ne prenant pas en compte les problèmes liés au partage de l'espace logique.

propriété d'*isolation* qui donne à chaque transaction l'illusion d'être la seule à s'exécuter. Dans cette exécution séquentielle équivalente, chaque donnée passe d'une valeur valide en début de transaction à une autre valeur valide à la fin de la transaction, en passant par des valeurs intermédiaires non valides. L'isolation peut être assurée à condition de prémunir les transactions de trois phénomènes : les lectures sales (une transaction ne doit pas lire une valeur intermédiaire produite par une autre transaction), les lectures non reproductibles (deux lectures d'une même donnée effectuées dans une même transaction doivent retourner la même valeur) et les modifications perdues (une version ne doit pas être ignorée par une transaction).

Se prémunir de ces trois phénomènes équivaut à construire le *graphe de dépendances* et s'assurer qu'il ne contient pas de cycles. Dans ce graphe une opération op_i dépend d'une autre opération op_j si elles portent sur la même donnée et que l'une au moins est une écriture ou qu'elles appartiennent à la même transaction. Cette dépendance implique que dans l'ordonnancement en série op_j doit précéder op_i. Or cela n'est possible que ce si op_j ne dépend pas également (directement ou indirectement) de op_i et, par conséquent, si le graphe est acyclique.

4.3.2.2 Sérialisation en mémoire centralisée

Les politiques permettant d'assurer la sérialisabilité en mémoire centralisée[6] sont connues sous le nom de *politiques de contrôle d'accès*. Cependant, puisque de nombreux principes sont repris dans le cas d'une mémoire répartie nous les confondons avec des politiques de cohérence et de synchronisation.

Afin de garantir la sérialisabilité des transactions, différentes approches ont été proposées, certaines reposant sur un contrôle continu de l'exécution des transactions, d'autres reposant sur un contrôle différé à la validation des transactions [BCF+97]. Les politiques à contrôle continu sont également appelées politiques pessimistes dans la mesure où elles supposent parfois à tort qu'une exécution concurrente potentiellement non sérialisable va forcément aboutir à une exécution non sérialisable et limitent ainsi les possiblités d'ordonnancement sérialisable.

Les plus connues et également les plus répendues des politiques à contrôle continu sont les politiques basées sur des techniques de verrouillage. Elles consistent à empêcher la formation de cycles en obligeant les transactions à protéger les données accédées par l'obtention préalable de verrous (verrou partagé pour une intention de lecture et verrou exclusif pour une intention d'écriture). Une transaction respectant

[6]Par mémoire centralisée nous entendons ici une mémoire où les pages logiques ne sont pas répliquées mais physiquement partagées par les transactions.

cette règle de protection est dite "bien formée". Une demande est conflictuelle si elle correspond à la demande d'un verrou exlusif sur une donnée déjà protégée par un verrou partagé ou exlusif ou bien à la demande d'un verrou partagé sur une donnée protégée par un verrou exlusif. Elle se traduit par la suspension de la transaction désirant accéder la donnée jusqu'à libération des verrous conflictuels. En outre, le principe de verrouillage à deux phases (Two-Phase -Locking ou 2PL) stipule qu'une fois qu'une transaction a libéré un verrou, elle ne peut en prendre un nouveau avant sa terminaison. Un ordonnancement de transactions concurrentes est alors sérialisable si chaque transaction est bien formée et respecte le principe du 2PL. Outre le fait que certains ordonnancements peuvent être rejetés à tort, les politiques par verrouillage doivent gérer les possibilités d'interblocage ou (appelés aussi "étreintes fatales") qui surviennent lorsque par exemple que deux transactions attendent mutuellement la libération d'un verrou détenu par l'autre. Pour régler ce problème certaines politiques augmentent le protocole classique afin d'éviter les possibilités d'interblocage tandis que d'autres utilisent des techniques de détection d'interblocage et détruisent une des transactions impliquées.

Une autre classe de politiques par contrôle continu regroupe les politiques par estampillage. Elles consistent à attribuer une estampille $ED(T)$ à chaque transaction T lors de son commencement et à associer à chaque page deux estampilles, une estampille $EL(P)$ égale à l'estampille de la dernière transaction ayant lue la page et une estampille $EE(P)$ égale à la dernière transaction ayant modifié la page. La méthode consiste à imposer que l'ordre des opérations effectuées sur une page respecte l'ordre d'estampillage des transactions. Une transaction T ne peut lire une page P que si $ED(T) \geq EE(P)$ et ne peut la modifier que si de surcroît $ED(T) \geq EL(P)$. Une optimisation de cette technique est connue sous le nom de la règle de Thomas qui consiste à ignorer les écritures conflictuelles qui sont aussitôt écrasées par une deuxième écriture avant qu'une transaction n'ait eu le temps de la prendre en compte. Bien que les problèmes d'interblocage ne se posent pas pour les politiques par estampillage, ces dernières imposent des dépendances entre les transactions dès le départ pouvant conduire, tout comme les politiques par verrouillage, au rejet d'ordonnancements sérialisables.

Les politiques optimistes par *certification* effectuent quant le contrôle de sérialisabilité lors de la phase de validation des transactions [KR81]. Pour cela, à chaque transaction sont associés deux ensembles correspondant respectivement aux pages qu'elle a modifiées et aux pages qu'elle a lues. Lors de la validation, le GMR vérifie que les opérations effectuées n'entrent pas en conflit avec les autres transactions. Selon la variante utilisée, la vérification est effectuée par rapport aux transactions validées

(certification en arrière), par rapport aux transactions actives[7] (certification en arrière) ou par rapport aux deux. L'ordre de sérialisation et ainsi déterminé par l'ordre de validation. Ce type de politique a la particularité de réduire considérablement le nombre d'interactions entre les traitements et le GMR mais nécessite cependant de maintenir une copie privée des pages modifiées et peut résulter en un encombrement important de la mémoire. De plus, du fait que la vérification est retardée, les transactions sont effectuées complètement même si leur première opération peut être la cause de leur rejet.

4.3.2.3 Sérialisation en mémoire répartie

Lorsque les données sont répliquées, assurer la sérialisabilité des transactions nécessite également de mettre les copies en cohérence. En plus de prévenir la formation de cycles, le GMR doit assurer qu'une transaction lit bien la dernière version des pages qu'elle accède. La notion de sérialisabilité est étendue à celle de une-copie sérialisabilité (one-copy serialisability) afin de prendre en compte ces propriétés [BHG87]. Cette notion permet de rendre transparente la réplication des données du point de vue des utilisateurs : une exécution une-copie sérialisable sur des données répliquées est équivalente à une exécution en série sur des données non répliquées.

Une taxinomie de différentes politiques basées sur du verrouillage assurant la une-copie sérialisabilité dans le contexte des bases de données à objets est présentée dans [Fra96][8]. Pour la plupart, ces politiques reprennent des idées développées pour des SGBD à disque partagé [Rah91]. L'auteur propose de distinguer essentiellement les politiques selon la manière dont elles préviennent l'accès à des copies invalides.

Politiques par évitement Les politiques par *évitement* préviennent l'accès à une valeur obsolète en respectant le principe "*read-one, write-all*" (ROWA) dont nous avons fait l'hypothèse dans le chapitre précédent. Ce principe stipule qu'à la terminaison d'une transaction ayant modifié une page P, toutes les copies de P ont la même valeur. Ce principe assure ainsi qu'une transaction lit toujours la dernière version d'une page et peut être mis en oeuvre soit en *invalidant* les copies non valides soit en *propageant* la nouvelle version d'une page. À l'inverse, les politiques par *détection* ne garantissent pas le principe ROWA et doivent détecter l'accès aux copies non valides.

[7] Une transaction active est une transaction en cours d'exécution mais non encore validée.

[8] Bien que les différentes propositions s'intéressent à la réplication entre différents clients d'un serveur, elles s'appliquent à notre contexte si l'on considère une architecture à la SHORE où chaque machine de la grappe se comporte à la fois comme un serveur et un client.

La politique Call-Back-Locking (CBL) fait partie de la première famille. Afin de réduire le nombre d'interactions avec le serveur, les verrous libérés par une transaction sont retenus sur sa machine d'exécution. De cette manière, les verrous sur les pages accédées par la transaction peuvent être obtenus localement (c.-à-d. sans interaction avec le serveur) par d'autres transactions s'exécutant sur la même machine. Ces verrous sont rapatriés par le serveur lorsqu'ils entrent en conflits avec des transactions distantes.

Cette politique se décline en deux variantes selon les types de verrous conservés sur les machines clientes et devant être rapatriés par le serveur lorsque nécessaire. La politique CBL-Read conserve uniquement les verrous en lecture tandis que la politique CBL-All conserve également les verrous en écriture. En conservant les verrous entre les transactions, cette politique tend à minimiser les synchronisations avec le serveur dont le coût est quasiment équivalent à lire une copie en mémoire distante. Cependant, en nécessitant une phase de rapatriement, cette politique tend à augmenter le temps d'obtention des verrous auprès du serveur et donc le temps de blocage. L'intérêt de garder un verrou dépend donc de la probabilité qu'une transaction locale demande ce type de verrou et de la probabilité qu'une transaction distante demande un verrou conflictuel.

La politique Optimistic-Two-Phase-Locking (O2PL) est une politique optimiste basée sur le verrouillage appartenant également à la famille des politiques par évitement. Tout comme pour la politique CBL, les pages résidantes peuvent être lues sans nécessiter de synchronisation immédiate avec le serveur. Les transactions sont autorisées à modifier leurs copies locales et ne demandent les verrous en écriture correspondant qu'à leur terminaison. Cette politique nécessite une forte réplication[9] et un cycle de dépendance se traduit, comme pour toute politique par verrouillage, par un inter-blocage.

Deux variantes de cette politique sont proposées différant selon la manière dont elles mettent en oeuvre le principe ROWA. La première consiste à invalider les copies qui contiennent des vieilles versions tandis que la deuxième consiste à les mettre à jour. La première permet de minimiser voire d'éliminer le coût de rapatriement induit par les verrous en écriture tandis que la deuxième permet de réduire le nombre d'interactions avec le serveur. Le choix de l'une ou l'autre dépend donc essentiellement des probabilités de réutilisation des pages et de conflit d'accès.

En ne vérifiant les conflits qu'à la terminaison de la transaction, cette politique

[9]Les auteurs supposent que chaque transaction possède son propre espace de travail (c.-à-d. son propre cache). Dans notre contexte, cela nécessite de répliquer les pages partagées par plusieurs transactions s'exécutant sur une *même* machine.

tend à minimiser le nombre d'interactions avec le serveur. En revanche, les inter-blocages sont décelés plus tardivement augmentant la perte de travail dans de tels cas et réduisant ainsi l'exploitation des ressources à des fins utiles. Comme toute politique optimiste, elle est donc surtout intéressante lorsque la probabilité de conflit est faible.

La politique AACC (Asynchronous Avoidance-Based Concurrency Control) [ÖVU98] se situe entre les politiques CBL et O2PL. Plutôt que d'obtenir les verrous en écriture de manière synchrone lors de l'accès ou à la terminaison de la transaction, la politique les demande au moment de l'accès de manière asynchrone. Les résultats montrent que cette politique apporte les avantages de la politique O2PL tout en détectant les cas d'inter-blocage bien plus tôt, réduisant ainsi le gaspillage des ressources.

Politiques par détection La politique par détection la plus simple est la politique par verrouillage Caching-Two-Phase-Locking (C2PL). Elle consiste simplement à conserver les copies des pages accédées localement afin de ne pas avoir à les récupérer auprès du serveur lors du prochain accès. Toutefois, la copie d'une page P peut ne pas représenter sa dernière version obligeant les transactions désirant l'accéder à s'assurer de sa validité auprès du serveur. Si la copie est valide seul le verrou est retourné et dans le cas contraire la nouvelle version de la page est également jointe. Cette politique ne permet donc pas de diminuer le nombre de blocages mais parvient à réduire l'encombrement du réseau et indirectement le temps d'attente d'une réponse du serveur.

Une extension de la politique précédente, appelée No-Wait-Lock (NWL), consiste à supposer de manière optimiste que les pages retenues sont valides et que les verrous demandés par une transaction seront accordés par le serveur [WR91]. Lors de l'accès à une page résidante localement, une transaction envoie sa demande de verrou et de mise en cohérence de manière asynchrone et continue son exécution. Si la page est invalide ou que le verrou ne peut être accordé à cause d'un inter-blocage, la transaction est détruite. Dans le cas contraire, le serveur ne fait rien et attend la demande de validation. Lorsque celle-ci arrive, le serveur attend que tous les verrous puissent être obtenus puis valide la transaction. Dans une variante (NWL-with-Notification), le serveur propage les nouvelles versions dans le but de réduire les risques de rejet des transactions.

4.3.2.4 Versionnement

L'intérêt des politiques de C&S basées sur du versionnement est d'accepter des exécutions qui sont rejetées par les politiques basées sur une version unique. Ces politiques sont particulièrement intéressantes lorsque de nombreuses transactions n'effectuent que des lectures et sont supportées par certains SGBD commerciaux. Elles reposent sur la notion de multi-version-sérialisabilité présentée en détail dans [BHG87].

Les politiques de C&S assurant la sérialisabilité que nous avons présentées préviennent les transactions des lectures "tardives" sur des pages modifiées. En conservant les anciennes versions des pages, il est possible d'autoriser ces transactions à lire une version valide qui n'est pas en cours de modification. Par exemple, avec la politique par estampillage présentée en 4.3.2.2, si $E(T_1) < E(T_2)$ et que T_2 a modifié la page P, toute tentative de lecture de P par T_1 conduit à son rejet (car elle essaie de lire une version créée dans son futur si l'on considère l'ordre de sérialisation). L'idée du versionnement et de conserver les dernières versions de P de manière à permettre à T_1 de lire la version qui lui est appropriée, c'est-à-dire la dernière validée avant son commencement. Les politiques de C&S par verrouillage et par certification peuvent également bénéficier de versions multiples. De telles politiques sont présentées et comparées dans [CM86]. Ces politiques distinguent les transactions modificatrices des transactions en lecture seule. En utilisant les versions, elles parviennent notamment à assurer la validation des transactions en lecture seule et à réduire considérablement le nombre de synchronisations.

Le coût de cette approche est cependant le maintien en mémoire des anciennes versions de P[10]. Toutefois, ce coût peut être amorti par la réplication nécessaire au partage entre les différentes machines. Par exemple, si les deux transactions T_1 et T_2 ne s'exécutent pas sur la même machine, T_1 pourrait éventuellement utiliser une version conservée dans sa mémoire locale ou dans la mémoire d'une machine autre que celle où s'exécute T_2.

Le système Eos adopte une telle approche en mettant en oeuvre la politique 2-Versions-2PL (2V2PL) [BP95]. La politique autorise simultanément un écrivain et plusieurs lecteurs : lorsqu'une page est demandée pour être modifiée, le système conserve l'ancienne version dans sa mémoire locale afin de satisfaire les demandes des lecteurs. Si cette approche ne permet pas d'éviter les inter-blocages elle assure cependant qu'une transaction en lecture seule est certaine d'être validée et ne sera pas bloquée par des transactions effectuant des modifications.

[10]Rappelons que dans notre contexte d'étude, l'interface du système de gestion de l'espace de stockage ne permet pas le versionnement des pages sur disque. Les versions que nous considérons ici sont "éphémères" (transient versionning) et correspondent à des copies ayant des valeurs différentes.

4.3.3 Sérialisation, performances et qualités de service

La sérialisation des traitements limite les ordonnancements possibles dans le but de garantir la propriété d'isolation des traitements. Cela a un impact sur les performances puisque certains traitements peuvent être bloqués voire même rejetés. Bien qu'un nombre important de politiques aient été proposées dans le but de minimiser les blocages en nombre et en temps, les contraintes d'ordonnancement peuvent aller à l'encontre d'autres qualités de services. De plus, la propriété d'isolation, très forte puisqu'elle ne fait pas d'hypothèses sur le comportement des traitements peut être relâchée afin d'obtenir des meilleures performances tout en assurant une qualité de service acceptable.

4.3.3.1 Gestion des priorités

L'interaction entre la synchronisation des transactions et la gestion de priorités a été largement étudiée dans les SGBD temps-réels. Dans ces systèmes, une transaction T_1 est plus prioritaire qu'une transaction T_2 si par exemple sa date limite d'échéance précède celle de T_2 ou si T_1 doit absolument finir avant T_2. Une manière simple d'atteindre ces objectifs est de privilégier T_1 quant à l'accès aux ressources matérielles (processeur, mémoire, disque, etc.) et donc de l'exécuter en priorité.

Les problèmes se compliquent cependant si T_1 et T_2 partagent des données et que l'intégrité de la base doit être préservée. La politique de C&S mise en oeuvre peut malheureusement être amenée à effectuer des choix d'ordonnancement allant en contradiction avec la gestion des priorités. Par exemple, si T_2 a lu une page P que T_1 désire modifier, une politique par verrouillage bloque T_1 jusqu'à la terminaison de T_2. Ce délai est d'autant plus long que la priorité de T_2 est faible par rapport aux autres transactions actives. Ce phénomène est appelé *inversion de priorités* puisqu'une transaction est exécutée avant une autre plus prioritaire. La conséquence est que les chances de terminer à la fois T_1 et T_2 avant échéance sont réduites et que terminer T_1 avant T_2 est impossible.

Pour cette raison, de nombreuses politiques de C&S prenant en compte les priorités des transactions ont été proposées dont quelques-unes sont présentées dans [Ald98]. Certaines d'entre elles sont basées sur la politique 2PL : 2PL-Wait-Promote (2PL-WP) et 2PL-High-Priority (2PL-HP). Lorsque la transaction T_1 se bloque en attente d'un verrou détenu par T_2, la politique 2PL-WP consiste à donner à T_2 la priorité de T_1 dans l'espoir de minimiser le temps de blocage de T_1. La politique 2PL-HP, au contraire, rejette et relance la transaction T_2 afin de permettre à T_1 de continuer son exécution.

Des politiques optimistes par certification pour lesquelles l'ordonnancement est décidé à la validation sont également présentées. La politique la plus notable est la politique OCC-Wait. Pendant la phase de validation, la transaction T_2 est comparée non pas avec les transactions déjà validées mais avec celles qui sont actives. Si un conflit est détecté avec une transaction T_1 plus prioritaire, la validation de T_2 est reportée après celle de T_1, la conduisant à un éventuel rejet. Ainsi une transaction ne peut pas être rejetée à cause d'une transaction moins prioritaire. De plus, en ordonnançant T_2 après T_1 il est possible que les deux transactions puissent être validées alors que la validation immédiate de T_2 entraîne forcément le rejet de T_1[11].

4.3.3.2 Sérialisation relâchée

L'isolation totale des transactions peut avoir de sérieuses conséquences sur les performances. Par exemple, une transaction T_1 constituée d'une seule requête effectuant une sélection portant sur un attribut non indexé nécessite la lecture de tous les n-uplets d'une relation. Afin de garantir la propriété d'isolation, T_1 doit obtenir un verrou partagé sur toutes les pages de la relation et les garder jusqu'à sa terminaison. En conséquence, une transaction T_2 désirant modifier un n-uplet de la relation doit attendre la terminaison de T_1 ce qui peut être long si la relation est grande. Ce qui est fâcheux ici est que T_1 n'a pas besoin de garanties aussi fortes que l'isolation. En effet, puisqu'elle ne lit chaque n-uplet qu'une fois, il n'est pas nécessaire de lui garantir que deux lectures d'une même page lui retourneront la même valeur. Ainsi, la plupart des systèmes commerciaux libèrent les verrous de T_1 aussitôt que les pages ont été lues, permettant ainsi à la transaction T_2 de ne pas attendre la terminaison de T_1.

Les niveaux d'isolation [GR93] ont été introduits pour relâcher le critère de la sérialisabilité jugé trop restrictif afin d'augmenter les possibilités d'ordonnancement tout en garantissant la cohérence du système. La norme SQL2 [IOfS92] définit les quatre degrés d'isolation suivants du plus fort au plus faible :

- Le niveau *Sérialisable* correspond à la sérialisabilité en comprenant les problèmes de fantômes.
- Le niveau *Lecture Reproductible* correspond au degré traditionnel d'isolation. Il assure qu'une transaction lisant plusieurs fois les même données lit toujours les mêmes valeurs mais n'est pas protégée contre l'apparition de fantômes.
- Le niveau *Lecture Valide* ne garantit plus la stabilité des lectures ni l'apparition de fantômes mais assure qu'une valeur lue correspond toujours à une valeur

[11] C'est le cas si par exemple si une P a été lue par T_1 et modifiée par T_2 et qu'aucun autre conflit n'existe.

produite par une transaction validée.

– Le niveau Lecture Dégradée n'assure aucune protection contre les problèmes de fantômes et les modifications non validées et n'assurent pas la reproductibilité des lectures.

Ce qui est intéressant est que ces conditions sont suffisantes pour assurer différents degrés d'isolation pour différentes transactions. Il est donc possible de supporter différents besoins et de limiter ainsi l'impact de l'isolation sur les performances.

Le parcours et la modification d'un index implanté sous la forme d'un B-arbre est un exemple typique nécessitant des garanties plus faibles que l'isolation de degré 3. Autoriser plus de parallélisme est d'autant plus important que certaines pages (notamment la racine) ont de fortes de chances d'être partagées. La recherche d'un n-uplet ne nécessite qu'une isolation de degré 2 puisque chaque page n'est lue qu'une fois. L'insertion et la suppression d'un n-uplet nécessitent cependant de protéger les noeuds modifiés par des verrous exclusifs. Une approche naïve consiste à poser des verrous exclusifs sur tous les noeuds et ne les relâcher qu'à la fin de la transaction. Cependant, le verrou exclusif protégeant le noeud racine exclu l'accès à l'index par toute autre transaction. Une autre approche consiste à ne pas respecter le principe du verrouillage à deux phases et effectuer un verrouillage couplé : les noeuds sont toujours verrouillés avant d'être lus ou écris mais un noeud est déverrouillé si la modification du noeud fils ne peut entraîner sa propre modification.

4.4 Conclusion

Dans ce chapitre nous avons fait l'hypothèse que les traitements n'ont pas forcément les mêmes comportements ni les mêmes exigences en terme de performance et de qualité de service. Dans la section 4.2, nous avons vu combien il est important de gérer explicitement le partage de la mémoire physique en la partitionnant et en implantant une politique d'allocation. De cette manière il est en effet possible d'isoler les traitements et de leur garantir un certain confort d'exécution. Nous avons vu également que l'allocation explicite permet d'informer les traitements sur leur espace d'allocation rendant possible leur adaptation.

Dans la section 4.3, nous nous sommes penchés sur les problèmes de partage de l'espace logique. Nous avons vu qu'il nécessite la mise en oeuvre d'une politique de cohérence et de synchronisation (C&S) dans le but de garantir un modèle de cohérence. Nous nous sommes particulièrement intéressés à la sérialisabilité des transactions, qualité de service garantie par les systèmes de gestion de bases de données. Nous

en avons vu les principes ainsi que différentes méthodes d'implantation en mémoire centralisée et en mémoire répartie. Nous avons ensuite parlé de l'impact d'une telle qualité de service sur l'ordonnancement des traitements et montré l'importance d'implanter des politiques capables de résoudre le phénomène d'inversion de priorités. Nous avons également dit quelques mots sur les possibilités de relâcher la sérialisabilité afin d'améliorer les performances lorsqu'une telle garantie n'est pas nécessaire.

La gestion du partage des espaces logique et physique est au coeur d'un serveur de données. Le choix des politiques de gestion est primordial si l'on désire atteindre de bonnes performances tout en garantissant un certain nombre de qualités de service. Il en découle que chaque type de serveur a ses propres exigences en termes de politiques et qu'il n'existe pas de politiques meilleures que les autres dans tous les domaines.

Chapitre 5

Support mémoire pour serveurs de données répartis

5.1 Introduction

5.1.1 Apports d'un support

Qu'un serveur de données soit utilisé pour gérer des comptes bancaires, supporter des applications de CAO ou multimédia, permettre l'extraction intelligente d'informations, qu'il ait des exigences faibles ou fortes en terme de cohérence, qu'il ait ou non des contraintes de type temps-réel, que sa préoccupation soit d'assurer un bon débit de traitements ou de minimiser leur temps de réponse, en bref quels que soient les services qu'il rend, les qualités de service qu'il garantit et les objectifs de performance qu'il vise, il doit dans tous les cas implanter des fonctions systèmes qui ont entre autres obligations d'autoriser :

1. la création et destruction de données dans un espace logique et leur associer une image permanente,
2. la lecture et la modification dans l'espace mémoire des données identifiées dans l'espace logique,
3. l'accès à un volume de données supérieur à la taille de la mémoire,
4. plusieurs traitements ayant des comportements différents à s'exécuter simultanément,
5. l'accès simultané à de mêmes données par différents traitements.

Une approche intéressante est par conséquent de regrouper ces fonctions communes sous la forme d'un *support mémoire*. La construction d'un serveur consiste alors à

utiliser ce support afin de mettre en oeuvre des fonctions de plus haut niveau telles que la gestion et l'exécution de requêtes, la gestion d'objets complexes, la gestion de présentations multimédia, etc. Offrir un support mémoire possède alors les avantages suivants :

Réutilisation L'isolation des fonctions de bas niveau du reste du code implantant le serveur permet leur réutilisation pour la construction d'un autre serveur. Leur conception et implantation une fois pour toutes permet ainsi de réduire le coût et le temps de travail nécessaires au développement de plusieurs serveurs.

Factorisation L'isolation de ces fonctions permet également leur partage par différents serveurs s'exécutant sur la même plate-forme. Il en résulte une réduction du code à charger en mémoire et une uniformisation de l'accès et du partage des ressources pouvant faciliter la cohabitation et la coopération des serveurs.

Facilité d'évolution Tant que l'interface du support ne change pas, l'évolution des fonctions de haut niveau est orthogonale à celle des fonctions de bas niveau. La modularité est reconnue depuis longtemps comme une méthode de conception facilitant l'évolution d'un logiciel.

Nous voulons dans ce chapitre présenter quelles fonctions doit implanter un support mémoire afin d'offrir les services décrits plus haut. Cet aspect-là est abordé dans la section 5.2 et nous montrons dans la section 5.3 comment les propositions existantes répondent à ces exigences.

5.1.2 Adaptabilité d'un support

La construction d'un support parfait capable de répondre aux exigences de tous les types de serveurs n'est, de notre point de vue, pas réalisable. Comme nous l'avons vu dans les deux chapitres précédents :

- le choix d'une politique de remplacement et de préchargement dépend des caractéristiques d'accès aux données et des informations statistiques disponibles et des objectifs de performance visés.
- le choix d'une politique d'allocation dépend des besoins en mémoire de chaque traitement, de leur différence de comportement et des différences de priorité qu'il peut exister entre eux.
- Le choix d'une politique de cohérence et de synchronisation dépend d'un compromis entre qualités de service souhaitées et performances et est dépendant d'autres contraintes telles que celles de type temps-réel.

Le choix d'une politique dépend des propriétés qu'elle garantit et des performances qu'elle apporte. Il nous semble que décider laquelle de deux politiques de gestion est la mieux adaptée à un contexte particulier peut être une tâche particulièrement difficile :

- Certaines études tendent à être trop généralistes. Les politiques sont certes efficaces pour de nombreux domaines d'application mais, ne parviennent pas à tirer le meilleur profit possible de contextes particuliers. Dans de nombreux cas, il est possible de tirer parti d'informations statiques ou dynamiques sur le comportement d'un traitement afin d'offrir une politique plus appropriée. Lorsque le comportement entre les traitements varient de manière significative, il semble plus intéressant d'offrir une politique locale à chaque traitement plutôt que d'implanter une politique globale.
- Il arrive que la comparaison d'une proposition avec d'autres soit incomplète. Même dans un domaine bien précis, avoir une connaissance exhaustive de tous les travaux similaires est difficile. De plus, une comparaison cohérente de deux propositions ne peut se faire que dans des conditions d'exécution équivalentes. Cette condition, qui n'est pas toujours respectée, nécessite souvent d'implanter des versions peu optimisées des algorithmes correspondant aux politiques comparées.
- Beaucoup de travaux étudient le comportement de leur politiques proposée en modélisant et simulant le contexte d'exécution. La simulation permet certes d'étudier le comportement d'une politique pour un large spectre de conditions d'exécution et de prédire son comportement dans le futur. Cependant, les modèles proposés simplifient la réalité en fixant, uniformisant ou ignorant certains facteurs. Le caractère imprévisible voire indéterministe des systèmes rend bien souvent ces simplifications peu réalistes.
- Étant donné l'incessante évolution des performances des différentes ressources matérielles et la manière dont les performances de ces ressources évoluent entre elles, les conditions d'étude d'une proposition peuvent vite s'éloigner de la réalité. Il est ainsi légitime de se demander si l'étude comparative de deux politiques menée il y a quelques années sont toujours aussi pertinente aujourd'hui.
- Les besoins et contraintes de gestion imposés par les serveurs sont sans cesse en évolution tant pour des raisons matérielles (minimisation du support matériel, utilisation de nouvelles technologies, etc.) que logicielles (création de nouveaux services, intégration de services, etc.). Ces besoins et contraintes peuvent nécessiter l'implantation de nouvelles politiques.

En conséquence, nous avons la conviction qu'il n'existe pas de politiques meilleures que toutes les autres. Nous pensons donc qu'un support doit être suffisamment adap-

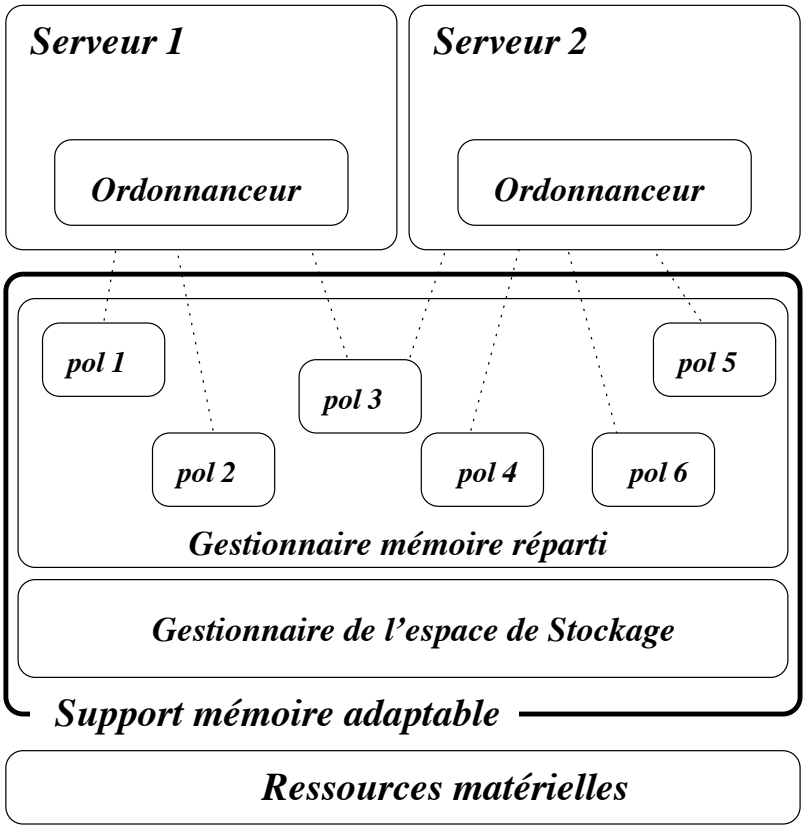


FIG. 5.1 – Support mémoire adaptable.

table de manière à laisser libre l'implantation de politiques de gestion prenant en compte les spécificités des serveurs à implanter. Cette approche a de plus l'avantage de pouvoir adapter la gestion mémoire d'un serveur en fonction de sa propre évolution. Modifier le comportement du support peut en effet permettre de prendre en compte au sein d'un même serveur de nouveaux besoins en terme de qualités de service et objectifs de performance, de changements dans le comportement des traitements ou dans l'environnement d'exécution, d'informations utiles jusqu'alors indisponibles, etc. La section 5.3 présente les principes généraux des systèmes adaptables dont nous présentons quelques exemples dans la section 5.4.

5.2 Architecture générale

La figure 5.1 illustre le partage d'un support mémoire adaptable par deux serveurs différents. Il implante un gestionnaire mémoire réparti (GMR) et un gestionnaire de l'espace de stockage (GES) qui assurent la gestion et l'accès aux ressources physiques et logiques pour le compte des deux serveurs. Toutefois, le GMR peut être adapté en

Nom fonction	arguments	résultat	description
créer_donnée	nombre_de_pages	id_logique	crée une donnée permanente de taille nombre_de_pages
détruire_donnée	id_logique		détruit une donnée permanente
début_traitement		id_traitement	déclare le début d'un traitement
fin_traitement	id_traitement		déclare la fin d'un traitement
début_accès	id_traitement, id_logique, num_page, mode		déclare le début d'un accès à une page logique et retourne l'adresse en mémoire où cette page peut être manipulée.
fin_accès	id_traitement id_logique, num_page		déclare la fin d'un accès à une page logique.

TAB. 5.1 – Interface minimale d'un support mémoire.

remplaçant certains de ses modules par des modules spécifiques à chacun des serveurs. Ces modules implantent des politiques de gestion qui correspondent aux besoins et comportement de chaque serveur ou bien des deux (politique *pol3*).

5.2.1 Interface utilisateur

Par support nous entendons une couche logicielle capable d'abstraire l'utilisation des ressources matérielles pour permettre à des traitements d'accéder aux données de l'espace logique. Il peut éventuellement intégrer d'autres couches logicielles déjà existantes mais cela est transparent du point de vue des traitements. Nous supposons qu'il offre un interface similaire à celle présentée dans le tableau 5.1. Cette interface, volontairement minimale, offre des points d'entrées vers les services minimaux que nous attendons d'un serveur de données. Notons qu'elle ne permet pas d'indiquer les qualités de service et objectifs de performance désirés. Au contraire, nous supposons que l'identité de l'appelant implicitement passé au support lui permet de déterminer ses besoins et son comportement et donc de décider de la manière dont doivent être gérées les ressources qui lui sont allouées.

5.2.2 Fonctions à mettre en oeuvre

Le tableau 5.2 résume les fonctions internes qui nous semblent devoir figurer dans un support mémoire. Comme le montre l'interface, le support doit permettre aux traitements de manipuler les données à l'aide de leurs identifiants logiques. La gestion de l'allocation dans cet espace (création et destruction de données) doit être réalisée

espace logique	EL1	création/destruction de données dans l'espace logique
	EL2	unicité des identifiants
	EL3	permanence des identifiants
	EL4	politique de cohérence et synchronisation
espace de stockage	ES1	création/destruction d'images permanence des données
	ES2	correspondance page logique / page disque
	ES3	permanence de la correspondance
espace physique	EP1	correspondance page logique / page physique
	EP2	politique de remplacement
	EP3	politique de préchargement
	EP4	politique d'allocation

TAB. 5.2 – Fonctions internes à implanter dans un support mémoire.

de manière transparente par le support (EL1) et chaque donnée doit être identifiée de manière unique (EL2). Les données pouvant être permanentes, l'état d'allocation de l'espace logique doit l'être aussi (EL3). Dans le cas contraire deux données créées dans deux sessions différentes pourraient avoir le même identifiant.

Le support doit pouvoir créer et détruire des images permanentes des données (ES1). Cette fonction est réalisée par le gestionnaire de l'espace de stockage (GES) dont nous avons supposé l'existence dans les chapitres précédents. Dans le contexte d'une grappe de machines, celui-ci doit être capable d'accéder les différents disques disponibles sur la plate-forme. Nous ne nous intéressons en effet pas ici aux problèmes du placement de données sur disque. De ce fait, l'interface minimale ne permet pas d'expliciter où doit être créée l'image permanente d'une donnée. Le support doit cependant pouvoir créer une donnée sur un disque quelle que soit la machine sur laquelle la demande de création est effectuée. Lorsqu'un traitement désire accéder une page non résidante, le GES doit être capable de retrouver la page disque correspondante. Cette correspondance doit donc être réalisée de manière interne et transparente du point de vue des traitements (ES2). Elle doit de plus être gérée de manière permanente afin de supporter l'arrêt du support (ES3).

Les données étant accédées en mémoire physique, le support doit assurer la correspondance entre une page logique et les pages physiques (emplacements) contenant une copie de la page (EP1). Comme nous l'avons fait remarquer en introduction du chapitre 3 une page logique peut être répliquée. Lorsque les pages sont répliquées sur une même machine, le support doit à partir de l'identifiant de l'appelant être capable de retrouver la copie concernée.

En plus des mécanismes que nous venons de décrire, le support doit mettre en

oeuvre les politiques des gestion dont nous avons parlées dans les chapitres précédents : politique d'allocation (EP2), politique de remplacement (EP3), politique de préchargement (EP4), politique de cohérence et synchronisation (EL4).

5.3 Adaptabilité

5.3.1 Principes

Les support ouverts, à l'inverse des supports monolithiques, présentent une architecture qui permet leur modification en vue de répondre à des besoins particuliers. Ceux-ci peuvent se présenter sous diverses formes :

- Un support est *extensible* si il offre la possibilité d'étendre les services qu'il met à disposition des utilisateurs. Les fonctions déjà existantes ne sont pas affectées par l'extension du système et leur comportement ne peut pas être modifié.
- Un support est *configurable* si il permet de spécifier l'utilisation d'une fonction particulière parmi un ensemble de fonctions disponibles dans le système. Lorsqu'une telle fonction correspond à une politique de gestion, il est possible d'infléchir le comportement du système sans toutefois pouvoir définir de nouveaux comportements.
- Un support est *adaptable* si les fonctions internes déjà présentes dans le système peuvent être remplacées par de nouvelles fonctions. En remplaçant les fonctions correspondant à des politiques de gestion il est possible de modifier le comportement du système afin de supporter de nouveaux besoins et contraintes de gestion.

Les systèmes qui nous intéressent sont bien entendu les supports adaptables puisqu'ils permettent l'ajout, le retrait et le remplacement des politiques de gestion qu'ils mettent en oeuvre. Dans un tel support il nous semble intéressant de bien distinguer les fonctions selon qu'il s'agit de politiques ou de mécanismes de gestion. Rappelons qu'une politique est une fonction susceptible de prendre des décisions en réaction à des changements d'état du système et dont le comportement dépend des qualités de service et objectifs de performance visés tandis qu'un mécanisme est une fonction permettant d'agir sur l'état du système et dont les choix d'implantation sont indépendants du type de serveur implanté.

5.3.2 Adaptabilité et continuité de service

Adapter un système nécessite l'ajout et le retrait de *modules* de code ainsi que leur connexion/deconnexion avec les autres modules. Offrir un support nécessitant une recompilation afin d'y intégrer de nouveaux modules n'est à notre avis pas suffisant. Son adaptabilité nécessite en effet l'arrêt du serveur le temps d'installer sur la plateforme la nouvelle version compilée du serveur. Il en résulte ainsi une indisponibilité des services qu'il offre pendant une période relativement longue qui peut être inacceptable pour les utilisateurs. L'ajout de modules doit donc pouvoir être effectué de manière *dynamique* sans nécessiter l'arrêt du serveur.

Garantir la continuité de la globalité des services offerts n'est cependant pas réaliste. Un module retiré ne peut en effet plus assurer les fonctions qu'il assurait auparavant. Même s'il est retiré pour être remplacé par un autre module assurant les mêmes fonctions, ces dernières sont indisponibles le temps du remplacement. Le remplacement d'un module et par conséquent d'une politique a donc des conséquences inévitables sur la disponibilité des services qui en dépendent. Il est cependant important de limiter cet impact aux seuls services concernés. Par exemple, si deux politiques de remplacement gèrent de manière indépendante l'espace d'allocation de deux traitements différents, changer une des deux politiques ne doit avoir de conséquences que sur l'exécution du traitement concerné. Cette opération doit être complètement transparente pour l'autre traitement.

5.3.3 Adaptabilité et sécurité

Les supports sont généralement des systèmes qui ont été testé, vérifiés voire prouvés afin de garantir leur bon fonctionnement. Introduire de nouveaux modules modifie leur comportement et peut avoir des conséquences désastreuses si les modules ajoutés sont défectueux ou mal intentionnés. Une première solution est de limiter les droits de modifier un support à des utilisateurs privilégiés tels que l'administrateur du serveur. Ce dernier prend la responsabilité d'ajouter des modules dont il est sensé avoir vérifié le bon comportement.

L'utilisation de langages orientés-objets (C++, Modula-3, Ada95, Java, etc.) permet la construction de programmes relativement bien structurés et réduit ainsi la probabilité d'erreurs. La gestion contrôlée de l'allocation et de la destruction des objets par l'intermédiaire d'un ramasse-miettes, et la restriction des opérations sur les références permet de plus de limiter les risques d'accès à des zones mémoire non valides. La génération automatique de code à partir de spécifications où des méthodes de vérification de programmes (telles que le *sandboxing*) permettent également de

garantir certaines propriétés comportementales. L'utilisation d'un interpréteur offre également un moyen de maîtriser l'exécution des modules ajoutés au support. Enfin, des mécanismes de protection offerts par le matériel peuvent également être utilisés afin de protéger les modules les uns des autres.

5.4 Propositions existantes

Cette section présente différents systèmes susceptible d'offrir un support mémoire adaptable pour la construction de serveurs de données répartis. Tous les systèmes offrant un certain degré d'adaptabilité ne sont pas représenté. En effet, le nombre de propositions de systèmes extensibles ou adaptables ne cesse de croître depuis quelques années. Une liste relativement complète de tels systèmes peut cependant être obtenue à l'adresse `http://www.cs.arizona.edu/people/bridges/oses.html`. Compte tenu de la diversité des propositions existantes, nous présentons celles qui nous semblent les plus utilisées et les plus intéressantes. Nous nous efforçons de montrer comment elles remplissent les conditions requises pour faire office de support mémoire ainsi que leur capacité d'adaptation.

5.4.1 Le système d'exploitation Unix

La plupart des serveurs de données utilisent les services offerts par un système d'exploitation. Un système d'exploitation de type Unix offre en effet de nombreux services et abstractions réduisant considérablement le coût d'implantation d'un serveur et permet l'exécution simultanée de différents serveurs.

La figure 5.2 montre comment un tel système peut être utilisé de manière à offrir un support mémoire tel que nous l'avons défini dans la section 5.2. La mémoire virtuelle, abstraction offerte par le système, peut être utilisée afin d'établir de manière totalement transparente et efficace la correspondance entre des pages logiques (pages virtuelles) et des pages physiques (EP1). Le système de gestion de fichiers permet ensuite de créer des images permanentes des données (fichiers) et l'interface `mmap()` permet de faire correspondre les pages disques aux pages virtuelles (ES1). Cette approche impose cependant d'utiliser la mémoire virtuelle comme espace logique. Le système est certes capable de gérer l'allocation de données en mémoire virtuelle et d'assurer leur unicité (EL1 et EL2) mais pas de manière permanente. L'état d'allocation de la mémoire virtuelle est en effet volatile et ne résiste à l'arrêt du système. La condition EL3 n'est donc pas vérifiée. De plus, le système est incapable de se souvenir après un arrêt à quelle adresse a été "mappé" un fichier. La correspondance entre une

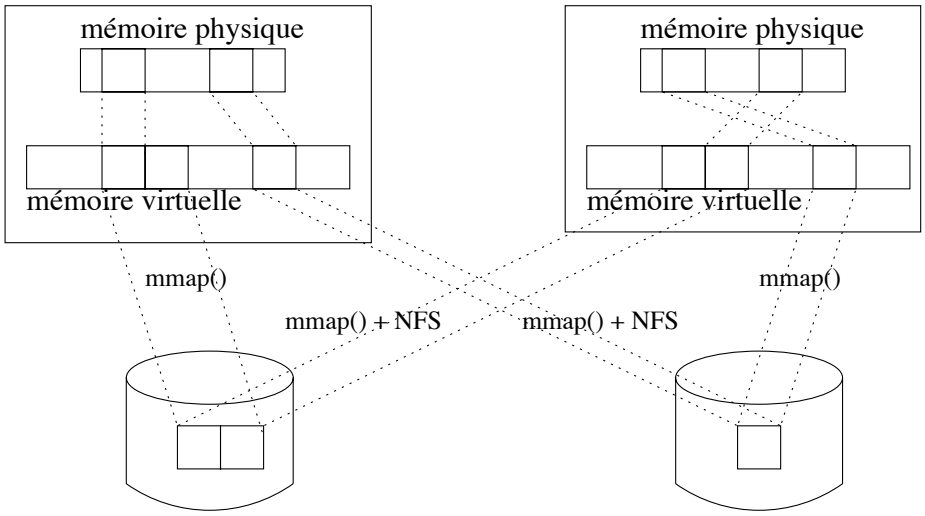


FIG. 5.2 – Utilisation des fonctions Unix pour implanter un support mémoire.

page disque et logique n'est donc pas non plus permanente (ES3).

Les problèmes se compliquent si l'on désire offrir un support *réparti*. Certes l'utilisation de NFS permet de partager des fichiers par les différentes machines et de faire correspondre une même page disque à différentes pages virtuelles. Cependant, chaque machine gérant sa propre mémoire virtuelle, une même page peut être identifiée par deux adresses virtuelles différentes. Dans un contexte réparti, la condition EL2 n'est donc pas remplie.

Les systèmes d'exploitation Unix implantent généralement la politique de remplacement LRU et la politique d'allocation working-set. La politique de préchargement OBL est également généralement implantée. Comme nous l'avons vu dans les chapitres précédents, ces politiques sont relativement généralistes et ne sont pas adaptées à certains comportements. Certains systèmes offrent cependant l'interface `madvise()` qui permet d'indiquer la manière dont les données comprises dans un intervalle de mémoire virtuelle vont être accédées. Si cette fonction offre un degré de configurabilité, les patrons d'accès sont cependant très limités et ne permettent pas d'informer de manière précise la gestion mémoire du système. De plus, beaucoup de systèmes offrent cette interface pour des raisons de compatibilité mais ne l'implantent pas réellement. De plus, les politiques sont appliquées localement à chaque machine. Unix est donc incapable de son propre chef de profiter des ressources réparties offertes par une grappe de machines. La politique de cohérence et synchronisation est celle implantée par NFS. Cette politique offre un modèle de cohérence relativement faible en assurant la visibilité des modifications à un intervalle de temps près.

5.4.2 La MVRP Munin

Une Mémoire Virtuelle Répartie Partagée (MVRP) est une abstraction permettant à des applications réparties sur différentes machines de partager des données en partageant un même espace d'adressage. L'abstraction offerte est la même que la mémoire virtuelle d'un système mais cette fois-ci l'allocation est gérée de manière répartie. Les MVRP mettent en oeuvre des mécanismes permettant de localiser les différentes copies d'une même page logique et offrent des modèles de cohérence beaucoup plus évolués que ceux offerts par le protocole NFS. Ici encore, l'interface `mmap()` permet de faire correspondre des pages logiques et des pages disques. Cependant, la plupart des MVRP sont volatiles : l'état d'allocation de la mémoire virtuelle et la correspondance entre l'espace logique et l'espace de stockage ne sont pas conservés de manière permanente.

Les politiques de remplacement mis en oeuvre pour ce type d'abstraction peuvent tirer profit de la mémoire distante. Malheureusement, la plupart des implantations font l'hypothèse que les traitements s'exécutent totalement en mémoire. La mémoire distante est uniquement utilisée pour des copies de pages partagées et non pour préserver des copies remplacées localement. En conséquence, les politiques de remplacement, de préchargement et d'allocation sont celles du système d'exploitation.

Les MVRP proposées dans la littérature ne sont pas généralement adaptable. À notre connaissance, seule la MVRP Munin [BCZ91] met en oeuvre différentes politiques de cohérence et de réplication. Les programmes ont la possibilité de choisir telle ou telle politique selon la manière dont les données sont partagées dans le but de minimiser la latence des opérations d'écriture et de lecture ainsi que de réduire l'encombrement du réseau. Les pages essentiellement lues sont répliquées tandis que celles essentiellement modifiées ont une copie unique qui est transférée de mémoire locale en mémoire locale au fur et à mesure des écritures. Les pages qui sont modifiées simultanément par plusieurs applications sont mises en cohérence lors d'opérations de synchronisation avec une politique de cohérence au relâchement. Munin n'est cependant pas un système adaptable dans la mesure où il n'est pas possible à une application de définir sa propre politique de cohérence mais simplement de choisir parmi un ensemble de politiques pré-définies.

5.4.3 Le noyau transactionnel Shore

Afin de faciliter et réduire le coût d'implantation de serveurs de systèmes de gestion de bases de données (SGBD) des noyaux transactionnels ont été proposés. Ils permettent à des traitements de créer des données et de les accéder de manière

transactionnelle en garantissant la sérialisabilité de ces traitements. Cependant, la plupart des noyaux transactionnels ont été proposés dans un contexte centralisé. Le plus représentatif des noyaux transactionnels réparti est le système Shore [CDF+94] qui présente une architecture client/serveur répartie.

Shore offre la possibilité de créer des objets permanents auxquels il attribue un identifiant unique et permanent (EL1, EL2, EL3, ES1). Il assure lui-même la correspondance (permanente) entre les pages logiques et les pages disques (ES2, ES3) et gère un cache dans lequel sont créées les copies des pages accédées (EP1). Il met en oeuvre une politique de remplacement mais pas de politique de préchargement ni d'allocation. La politique de remplacement, qui est décrite dans la section 3.3.2, essaie de réduire le plus possible la réplication entre les différentes machines. Il met en oeuvre la politique de cohérence et synchronisation CBL décrite dans la section 4.3.2.3.

Shore offre ainsi un très bon support pour la construction de serveurs de données. Il n'est cependant et malheureusement absolument pas adaptable. Les politiques implantées sont fixées et ne peuvent donc pas être adaptées pour des conditions d'utilisation particulière. Augmenter le système afin d'offrir des garanties de types temps-réel nécessiterait une modification significative du support.

5.4.4 Le micro-noyau Mach

La philosophie des micro-noyaux est de déplacer le plus d'abstractions et de services possibles de l'espace noyau vers l'espace utilisateur. Comme le montre la figure 5.3, les fonctions de haut niveau, comme par exemple les systèmes de gestion de fichiers, sont implantées sous la forme de *serveurs*, c'est à dire un processus utilisateur avec un espace d'adressage propre. Attribuer un espace d'adressage utilisateur à chaque serveur protège les serveurs les uns des autres et les protège des applications. De plus, cette approche permet de protéger de manière radicale le noyau des serveurs. Ainsi, un serveur défectueux ou mal intentionné ne peut en aucun cas causer de torts à d'autres serveurs.

Notons qu'un des avantages des micro-noyaux est d'être généralement conçus de manière à être facilement portés sur différentes architectures matérielles. Il permettent ainsi d'offrir une couche logicielle homogène sur une grappe de machines hétérogènes. Le principal inconvénient de cette approche est cependant le coût des communications (IPC) entre les serveurs et les applications qui nécessitent pour chaque appel plusieurs changements de contexte.

Parmi les micro-noyaux les plus connus et utilisés nous trouvons Mach [ABB+86] et Chorus [BGG+91]. Tous deux offrent la possibilité de définir au niveau utilisateur un

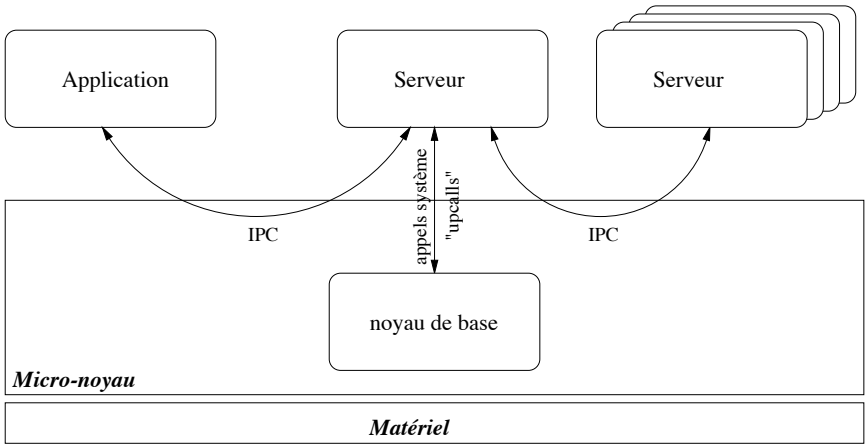


FIG. 5.3 – Architecture d'un micro-noyau.

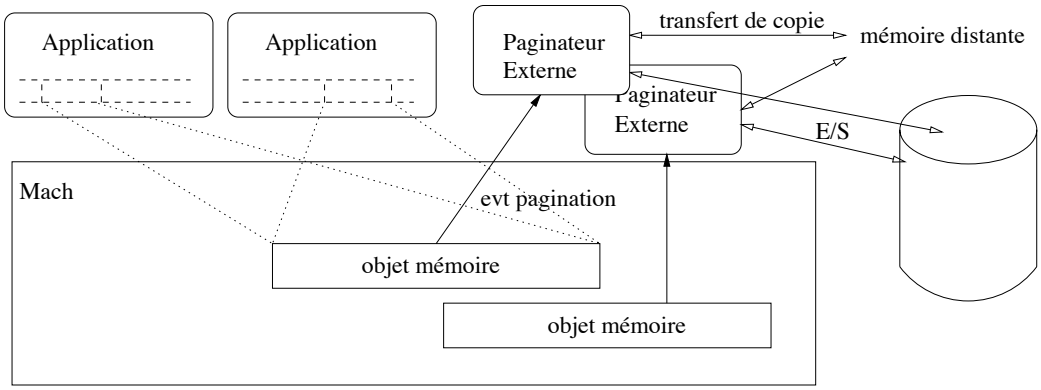


FIG. 5.4 – Paginateur externe dans Mach.

paginateur externe (external pager). Comme le montre la figure 5.4, sa responsabilité est de gérer la pagination, c'est-à-dire les échanges de pages entre la mémoire physique et l'espace de stockage. Lorsqu'un défaut de page est détecté, le système informe le paginateur de l'adresse virtuelle de la page manquante. Ce dernier charge la page et informe le système que la page est chargée. De même, lorsque le système décide de remplacer une page, le paginateur en est informé et peut décider d'écrire la page sur disque. Plusieurs paginateurs peuvent être définis mais sont alors responsables de la pagination d'*objets-mémoires* différents. Un objet-mémoire représente un ensemble de pages ayant une représentation en mémoire physique, sur disque ou ailleurs et pouvant être couplé à différentes adresses dans l'espace virtuel des processus (i.e. segment).

Le paginateur externe permet ainsi le contrôle de la correspondance entre l'espace logique et l'espace de stockage. Cette faculté est notamment cruciale lors de la mise en oeuvre d'un serveur de données transactionnel dont les mécanismes de résistance aux pannes ont une impact sur le moment où les données peuvent être écrites dans la

base [GR93]. Par exemple, l'utilisation d'un journal de type "image après", empêche de reporter dans la base les modifications effectuées par une transaction avant la validation de cette dernière (politique No-Steal). Grâce à cela, Mach a pu être utilisé comme support des SGBD Cricket [SZ90] et Camelot [SPB88].

Le paginateur externe offre la possibilité de préserver les pages remplacées en les transférant en mémoire distante. Inversement, sur un défaut de page, le paginateur peut charger la page à partir d'une autre machine au lieu de la lire sur disque. Toutefois, Mach et Chorus n'offrent pas de mécanismes de localisation des différentes copies d'une page. De plus, la politique de remplacement reste sous le contrôle du noyau et il n'est donc pas possible d'implanter une véritable gestion de la mémoire globale.

Mach tout seul n'offre pas un support mémoire tel que défini dans la section 5.2. Son extensibilité permet toutefois d'implanter les fonctions nécessaires à un support mémoire (EL2 à EL4, ES1 à ES3, EP2[1] et EP3). Il n'est cependant pas possible de modifier la politique d'allocation du micro-noyau et la mise en oeuvre d'une politique de remplacement répartie nécessite l'existence d'un mécanisme de localisation.

5.4.5 Le système extensible Spin

Spin [BCE+94, SFPB95] est un système d'exploitation extensible qui permet l'adjonction dynamique de code dans le noyau. La protection du noyau dans Spin est assurée en imposant le langage de programmation Modula-3 dont les caractéristiques (modularité, encapsulation, vérification de type, allocation mémoire automatique) empêchent les extensions d'accéder directement aux ressources du système et les isole les unes des autres.

La gestion de la mémoire virtuelle dans Spin est séparée en trois composants : la gestion des pages physiques, la gestion d'allocation d'adresses virtuelles et l'association entre pages virtuelles et pages physiques. L'interaction entre les systèmes et les extensions est essentiellement de type événement/réaction. Concernant la gestion de la mémoire, une extension peut réagir aux événements de défaut de page et de remplacement concernant des objets mémoires. Spin a ainsi pu servir de support au noyau transactionnel Rhino qui implante sa propre politique de remplacement et de pagination en accord avec la politique de journalisation utilisée [Sai97]. Spin gère l'allocation de pages physiques entre les différents objets-mémoires en implantant une politique globale de remplacement de type FIFO. Le système transmet sa décision à l'extension qui gère l'objet mémoire concerné (par exemple Rhino), lequel peut choisir une autre victime parmi les pages des objets mémoires qu'il gère.

[1] En supposant l'architecture proposée dans [MA90].

Les apports de Spin en tant que support sont ceux d'un système d'exploitation dont la gestion du remplacement et de la pagination est adaptable. Il ne s'agit cependant pas d'un système réparti et ne réalise en conséquence pas les conditions EL2, EL3, EL4 et ES3.

5.4.6 Premo

Une modification de Mach permettant de d'adapter la politique de remplacement proposée dans [MA90] sous le nom de Premo. Dans la nouvelle architecture proposée le paginateur est responsable non seulement de la pagination des objets-mémoires auxquels il est associé mais également de la politique de remplacement. Lorsque le noyau a besoin de récupérer un emplacement, il sélectionne la page la moins récemment utilisée (LRU). Si aucun paginateur n'est associé à l'objet mémoire auquel elle appartient, la page est supprimée. Dans le cas contraire le micro-noyau demande au paginateur correspondant de libérer un ou plusieurs emplacements. La politique LRU appliquée par le noyau de manière globale est ainsi utilisée comme politique d'allocation entre les différents objets-mémoires tandis que les paginateurs peuvent décider du remplacement des pages d'un ou plusieurs objets-mémoires.

5.4.7 Hipec

Le système Hipec [LCC94] propose un ensemble d'instructions permet de définir des politiques de remplacement. Ces instructions permettent l'exécution de commandes simples (opération sur des entiers, comparaisons, branchements) ou plus complexes (opérations sur des listes ou sur des caches) et sont interprétées par le noyau qui garde ainsi un contrôle total de l'exécution du programme. Hipec adopte une approche consistant à fournir un langage interprété pour l'implantation de politiques de remplacement. Une application peut décrire l'algorithme de la politique dans ce langage et l'associer à une région mémoire. Lorsque le système décide de remplacer une page, il charge le code dans le noyau et l'interprète. L'interprétation permet au système de protéger complètement les autres ressources du système d'un mauvais comportement de la politique.

La politique doit définir deux procédures, l'une activée sur un défaut de page, l'autre activée lors d'un remplacement. Le langage fournit des instructions arithmétiques et logiques, de manipulation de files de pages (insertion, suppression, recherche, appartenance, etc.), de consultation et modification de l'état d'une page (bits de modification et de référence), d'interaction avec le système (allocation, libération), etc.

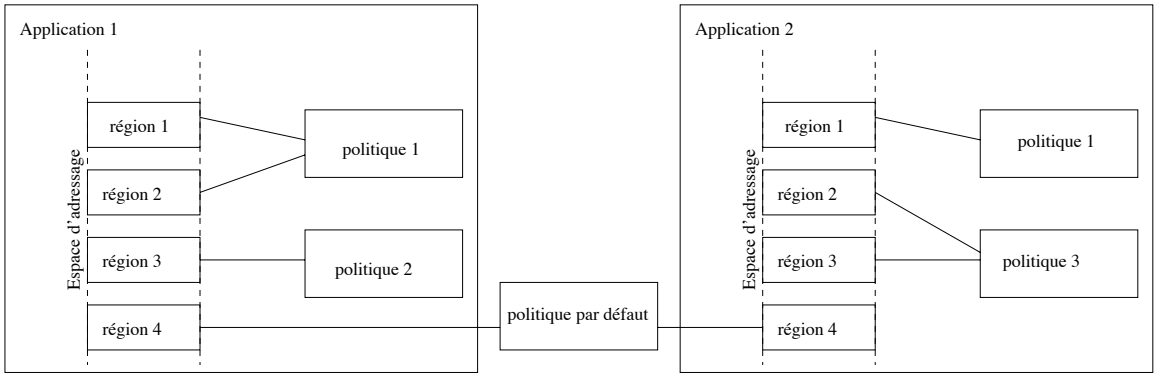


FIG. 5.5 – Association des politiques de remplacement dans Hipec.

permettant de définir des politiques spécifiques, ainsi que des instructions permettant d'appliquer des politiques de gestion pré-définies (FIFO, LRU, MRU).

Contrairement à l'approche prise pour Premo ou les politiques sont attachées à des objets mémoire éventuellement partagés par plusieurs applications, l'approche prise ici consiste à associer une politique à une région de l'espace d'adressage d'une application (cf. figure 5.5). Si cette approche permet de définir différentes politiques pour différentes régions d'une même application, elle ne permet cependant pas de définir une politique de remplacement pour des données partagées.

La politique d'allocation employée par le noyau divise statiquement la mémoire en deux partitions de tailles égales, l'une pour les applications ayant leurs propres politiques de remplacement, l'autre pour les applications utilisant la politique de remplacement par défaut (LRU). Lorsqu'une application déclare sa politique, elle passe en paramètre le nombre d'emplacements minimum nécessaire à la gestion de la région concernée. Si ce nombre ne peut être garanti, l'application est suspendue jusqu'à ce que sa requête puisse être satisfaite. A l'exécution, lors de défaut de page, la politique peut réclamer des emplacements supplémentaires. Les emplacements supplémentaires sont ainsi alloués aux premiers demandeurs. Une politique FIFO est utilisée pour sélectionner l'application qui doit rendre en premier des emplacements.

Si cette approche résout complètement les problèmes de protection tout en réduisant le nombre de traversées du noyau, elle hérite cependant des problèmes de performances spécifiques aux langages interprétés. De plus, l'algorithme de remplacement doit être chargé depuis l'espace utilisateur dans l'espace noyau à chaque défaut de page. Enfin, la politique d'allocation de l'espace physique entre application spécifiques et applications non spécifiques est très discutable.

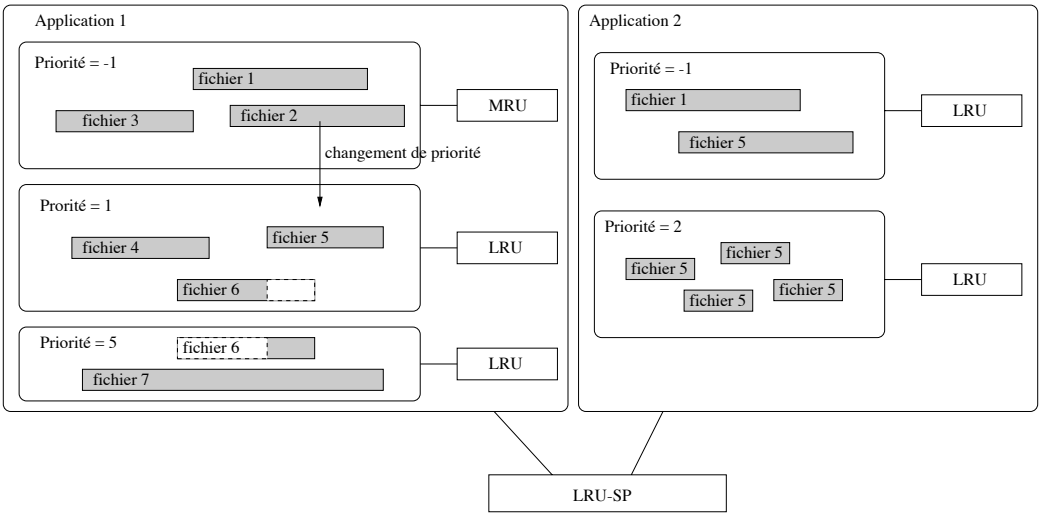


FIG. 5.6 – Allocation et remplacement dans [CFL94].

5.4.8 Cao et al

Dans [CFL94], les auteurs proposent de modifier Ultrix de manière à fournir une interface permettant à une application d'associer à l'exécution un niveau de priorité à un fichier et une politique de remplacement (parmi un ensemble de politiques pré-définies) à un niveau de priorité (cf. figure 5.6). Parmi les pages allouées à une application, le système remplace en premier les pages des fichiers qui ont la plus faible priorité et applique la politique de remplacement associée de manière globale à l'ensemble des pages d'un même niveau de priorité.

Le choix d'une politique de remplacement s'applique de manière globale à l'ensemble des fichiers ayant la même priorité. De cette manière, une application peut appliquer une même politique pour tous les fichiers, différentes politiques pour différents groupes de fichiers et changer les priorités afin que le système remplace les pages d'une fichier avant celles d'un autre. Afin de permettre un contrôle fin, une fonction supplémentaire permet en outre de modifier de manière temporaire la priorité d'un ensemble de pages d'un fichier. Lors d'un changement de priorité, les pages sont introduites dans la nouvelle politique de manière à être remplacées en dernier.

Ils proposent d'assigner un faible priorité et une politique MRU aux fichiers parcourus de manière séquentielle (ou séquentielle et cyclique), de regrouper les petits fichiers sous la même priorité de manière à avoir un nombre significatif de pages sous la même politique, d'assigner une forte priorité aux fichiers et d'assigner une faible priorité à un fichier dès qu'il n'est plus utile.

La politique d'allocation entre les différentes applications et la politique LRU-SP

	EL1	EL2	EL3	EL4	ES1	ES2	ES3	EP1	EP2	EP3	EP4
Unix	√			f	√	√		√	c	f	f
Munin	√	√		C	√	√		√	C	f	f
Shore	√	√	√	F	√	√	√	√	F	f	f
Mach	√							√	f	f	f
Spin	√				√	√		√	a	f	f
Premo	√							√	a	f	f
Hipec	-	-	-	-	-	-	-	-	a	-	-
Cao et al	-	-	-	-	-	-	-	-	a	-	-

F = fixe, C = configurable, A = adaptable, - : non évaluable
En minuscule : apport limité, en majuscule : apport significatif

FIG. 5.7 – Apport des systèmes existant en terme de support mémoire adaptable.

présentée au chapitre précédent. Rappelons qu'elle consiste à appliquer la politique LRU de manière globale afin de choisir une application devant rendre un emplacement, mais de laisser à l'application le choix de la page à remplacer. La gestion de la file LRU globale est telle qu'entre autre la décision d'une application ne peut être nuisible aux autres applications.

Bien que cette proposition permette de définir une politique de remplacement à deux niveaux et de changer de configuration en cours d'exécution, elle souffre d'un certain nombre de limitations. Tout d'abord elle ne s'applique qu'aux fichiers et non à la mémoire virtuelle et ne permet pas de définir de politiques spécifiques (seules les politiques LRU et MRU sont offertes). De plus, l'association entre niveaux de priorité et politique de remplacement ne permet pas de changer la priorité d'un fichier sans changer sa politique. Or le fait qu'un fichier ait moins de chance d'être accédé qu'auparavant n'implique pas forcément qu'il va être accédé différemment. Ensuite, la politique d'allocation intra-application offre des possibilités limitées tandis que celle inter-applications n'en offre aucune. Enfin, rien n'est dit concernant le comportement du système lors du partage d'un même fichier par deux applications différentes.

5.5 Conclusion

Nous avons vu dans cette section l'intérêt d'offrir un support mémoire pour la construction de serveurs de données (réutilisation, factorisation, facilité d'évolution). Nous avons également mis en avant la nécessité d'offrir un support adaptable c'est-à-dire dont le comportement peut être adapté pour des besoins et comportement particuliers. Nous avons caractérisé les conditions requises pour qu'un support ait

un apport suffisant pour la construction de serveurs de données et nous avons pu constater, comme le récapitule le tableau 5.7, que ces conditions ne sont remplies par aucune des propositions existantes.

Outre le fait que ces systèmes offrent des fonctions limitées pour la gestion de ressources logiques et physiques réparties, ils n'offrent pas un niveau d'adaptabilité suffisant compte tenu des possibilités de gestion d'une mémoire répartie :

- Les quelques propositions pour l'ajout de politiques de remplacement ne sont pas suffisantes car elles ne permettent pas d'isoler les comportements à la fois selon les données et selon les traitements ni d'utiliser la mémoire distante.
- Une même page ne peut être considérée que par une seule politique de remplacement même si elle a une utilité pour deux traitements différents.
- Aucune proposition pour adapter la gestion de l'allocation n'a été proposée.
- À part Munin, aucun système réparti ne permet de passer d'un modèle de cohérence à un autre. De plus, même dans Munin, il n'est pas possible de définir ses propres politiques de cohérence et synchronisation.
- Aucun support permettant l'ajout de politiques de préchargement spécifiques n'a été proposé.

En conséquence, aucun système n'offre selon nous un support suffisant pour faciliter la construction d'un serveur de données sur une architecture répartie. Partant de cette constatation, nous proposons dans le chapitre suivant ADAMS, un support mémoire adaptable offrant une base très attractive pour la construction de serveurs de données.

Chapitre 6

ADAMS, un support mémoire adaptable

6.1 Introduction

6.1.1 Motivation

Dans les chapitres 3 et 4 nous avons montré l'importance des politiques de remplacement, de préchargement, d'allocation et de cohérence et synchronisation (C&S). Malheureusement, parmi les supports qui ont été présentés dans le chapitre 5, peu permettent de spécifier convenablement des politiques de gestion particulières, encore moins d'en spécifier plusieurs en même temps et aucune ne le permet pour tous les types de politiques que nous considérons.

Ce deuxième point est particulièrement important car, indépendamment de leur type, toutes ces politiques sont concernées par des objectifs de performances et qualités de service, et dépendantes du comportement des traitements. Le gain qualitatif et quantitatif obtenu par la spécialisation d'un type de politique peut être négligeable s'il n'est pas possible de spécialiser les autres types de politique.

6.1.2 Objectifs

Ce chapitre présente une proposition de support mémoire dont l'objectif est :

- de faciliter la construction de serveurs de données répartis sur une grappe de machine et de faciliter l'évolution de leur gestionnaire mémoire,
- de permettre l'ajout et le retrait dynamique de politiques d'allocation, de remplacement, de préchargement, et de cohérence et synchronisation

- de séparer du mieux possible les responsabilités des différentes politiques tout en facilitant leur intégration,
- de supporter l'existence et la cohabitation de différents types de politiques et l'exécution de différents serveurs

Nous nous basons pour cela sur un modèle de gestion hiérarchique de la mémoire qui permet de définir différentes politiques pour différents contextes de gestion et à différents niveaux. Ce modèle de gestion se justifie par le fait que les serveurs exhibent généralement un profil d'exécution et de gestion hiérarchisé. De plus, la cohabitation de différents serveurs, services ou politiques peut nécessiter l'implantation d'une politique capable de gérer le partage des ressources physiques et logiques, et nécessite par conséquent une gestion hiérarchique de la mémoire.

La section 6.2 introduit la notion de contexte et présente les différents concepts du modèle. La section 6.3 présente l'architecture du support ADAMS et notamment les services pour la gestion des interactions de type notification/réaction. Enfin, la section 6.4 propose des interfaces et montre comment elles offrent un support pour les différents types de politique considérés. La section 6.5 conclue ce chapitre.

6.2 Modèle de gestion

6.2.1 Notion de contexte

Notre modèle repose sur le concept de *contexte* de gestion d'accès. Ce concept représente la manière dont sont gérés un ensemble d'accès et rassemble un ensemble de politiques responsables de la gestion de ces accès. Il nous permet d'isoler différents ensembles d'accès qui n'ont pas les mêmes comportements ou ne répondent pas aux mêmes contraintes de performance ou de qualité de service.

Un contexte peut par exemple représenter l'ensemble des accès effectués par les requêtes d'un serveur $S1$ sur une base de données regroupant un ensemble de relations indexées. Les politiques de gestion associées sont celles du gestionnaire de mémoire du serveur. Parallèlement, un autre contexte peut représenter les accès effectués par les requêtes d'un autre serveur $S2$ sur la même base de données. $S2$ possède ses propres politiques de gestion mémoire qui peuvent être différentes de celles de $S1$.

Bien souvent un contexte représente des accès qu'il est souhaitable de différencier. Par exemple, les serveurs $S1$ et $S2$ peuvent avoir des objectifs de performance et/ou qualité de services suffisamment différents pour justifier leur isolation. De même, les requêtes du serveur $S1$ peuvent avoir des différences de comportement et/ou besoin d'allocation nécessitant leur isolation. De ce fait, les contextes doivent pouvoir être

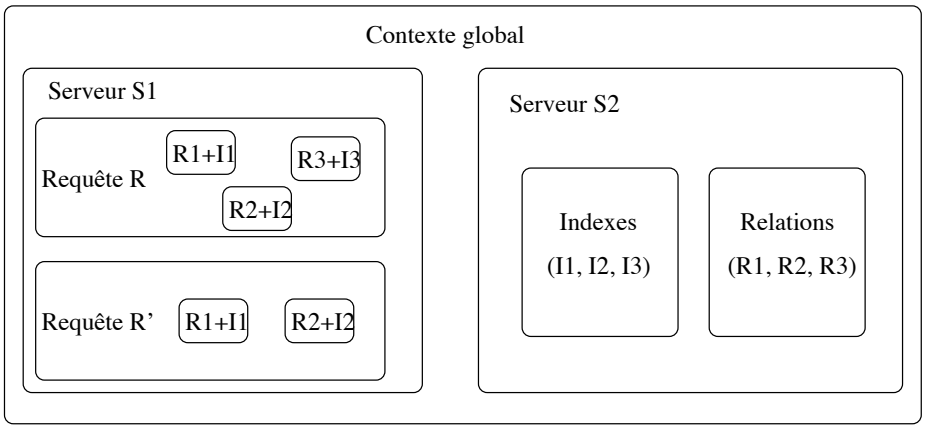


FIG. 6.1 – Exemple de hiérarchie de contextes (représentation ensembliste).

imbriqués, c'est-à-dire un contexte doit pouvoir inclure un ou plusieurs autres sous-contextes.

Dans l'exemple de la figure 6.1, le contexte global représente l'ensemble des accès effectués par tous les traitements et inclut les serveurs $S1$ et $S2$. À son tour, le serveur $S1$ inclut différents contextes afin de différencier les accès provenant de différentes requêtes. Enfin, les accès effectués par une même requête sont différenciés en fonction des relations qui sont accédées. Cette organisation correspond à celle proposée pour les politiques d'allocation entre différentes requêtes que nous avons présenté dans la section 4.2.3. Dans le contexte du serveur $S2$, les accès sont différenciés en fonctions du type des données qui sont accédées. Un contexte représente l'ensemble des accès effectués sur des index alors qu'un autre représente l'ensemble des accès effectués sur des relations. Cette fois l'organisation correspond à celle proposée en 4.2 pour la gestion du partage de l'espace physique par séparation des domaines.

L'ensemble des politiques de gestion associées aux différents contextes forme une gestion hiérarchique de la mémoire. Pour simplifier, nous dirons que les responsabilités d'un contexte sont celles des politiques qu'il regroupe. Dans notre exemple, le contexte global est responsable de la gestion du partage de l'espace logique : il décide quand un serveur peut accéder une page logique et quelle copie il est autorisé à lire ou modifier. Il est également responsable du partage de l'espace physique et décide de la distribution des emplacements entre les différents serveurs. À son niveau, le serveur $S1$ a pour tâche de synchroniser les contextes représentant les accès de ses requêtes R et R' sur les copies qui peuvent être lues ou modifiées. Il est en outre responsable de l'allocation mémoire entre les différentes requêtes. La distribution des emplacements alloués à une requête entre les différentes relations parcourues est laissée à la charge du contexte associé à la requête. Enfin, au niveau le plus bas, le contexte associé à un

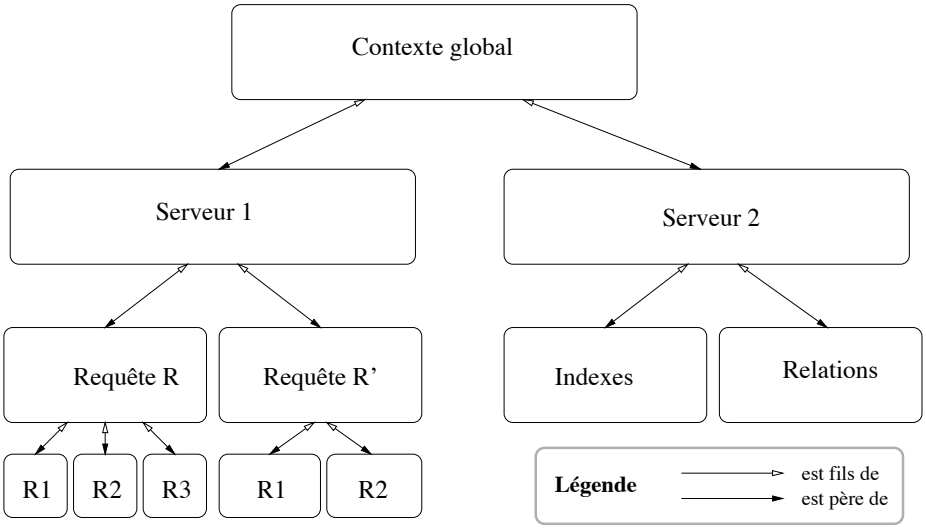


FIG. 6.2 – Exemple de hiérarchie de contextes (représentation hiérarchique).

parcours de relation est responsable du remplacement des copies de pages appartenant à la relation et accédées par la requête correspondante.

Les contextes regroupent ainsi un ensemble de politiques réalisant une gestion hiérarchique de la mémoire. Nous les représentons, de ce fait sous la forme d'un arbre dont les noeuds représentent des contextes intermédiaires, les feuilles des contextes terminaux et les arcs les relations d'inclusion ou de *parenté* entre les contextes. La figure 6.2 illustre cette représentation pour l'exemple considéré.

6.2.2 Formalisation des concepts

Les différents concepts intervenant dans notre modèle sont résumés dans le tableau 6.1 et leur schéma d'association est illustré par la figure 6.3.

Nous désignons par ET l'ensemble des traitements que nous assimilons à des flots d'exécution ou processus non répartis. Un traitement est identifié par le numéro de processus et du noeud sur lequel il s'exécute. L'espace logique, désigné par EL, est constitué de pages logiques de taille fixe et identifiables de manière unique. L'espace physique, noté EP, regroupe l'ensemble des emplacements des mémoires physiques de la grappe, un emplacement étant identifié par son adresse physique et le numéro de la machine à laquelle il appartient. Un accès est un couple $(page, traitement) \in EL \times ET$ et une copie est un triplet $(page, emplacement, version) \in EL \times EP \times \mathbb{N}$, l'espace des copies étant noté $\mathcal{C}$. Un emplacement ne contient qu'une seule copie, c.-à-d. $(p, e, v), (p', e, v') \in \mathcal{C} \Rightarrow p = p', v = v'$. La version constitue un moyen de différencier deux copies d'une même page ayant des valeurs différentes et non interchangeables.

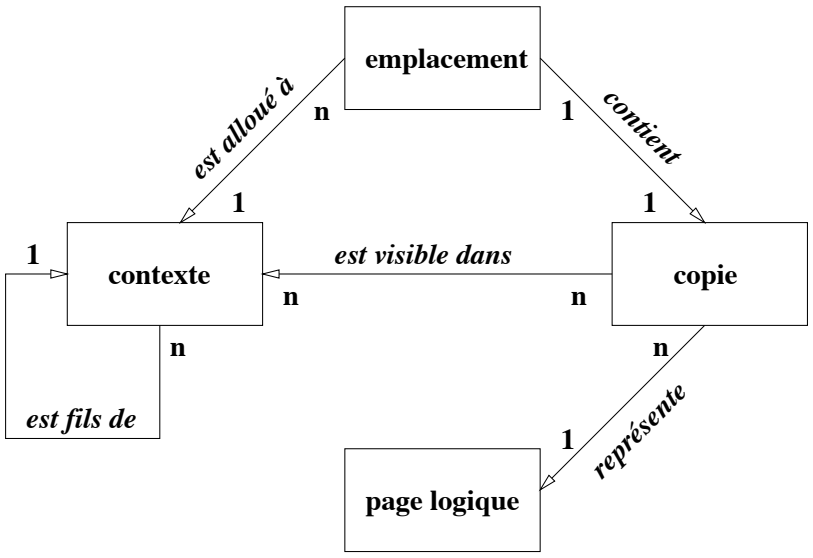


FIG. 6.3 – Schéma d'association entre les concepts du modèle de gestion mémoire hiérarchique.

Nous dirons que deux copies sont identiques si elles représentent la même page et ont la même version.

Afin de différencier les contextes, nous leurs associons un *espace de gestion*. L'espace de gestion d'un contexte C, noté $EG(C)$, est un ensemble de couples $(page, traitement) \in EL \times ET$ et représente les accès dont la gestion est (en partie) de sa responsabilité. Le père et les fils d'un contexte C sont respectivement désignés par $P(C)$ et $F(C)$. L'espace de gestion d'un contexte doit être inclus dans celui de son père et ceux de ses fils doivent être disjoints deux à deux, c'est-à-dire $EG(C) \subseteq EG(P(C))$ et $\forall C1, C2 \in Fils(C)$, $EG(C1) \cap EG(C2) = \emptyset$. Un contexte est intermédiaire s'il admet des fils et terminal sinon. Pour tout accès a il doit au moins exister un contexte terminal ctx tel que $a \in EG(ctx)$. Nous notons EC l'espace des contextes.

Un lien est l'association d'un contexte et d'une copie ou plus exactement un triplet $(contexte, copie, typelien) \in EC \times \mathcal{C} \times \{primaire, secondaire\}$. Un lien, primaire ou secondaire, indique que la copie peut être consultée et modifiée dans le contexte[1]. Un lien primaire indique de plus que l'emplacement contenant la copie lui est associé (c.-à-d. alloué). Comme le montre la figure 6.3, un emplacement ne peut être associé qu'au plus à un contexte et il existe donc au plus un lien primaire vers une copie. Ces deux types de lien permettent d'exprimer le partage physique d'une copie dans différents contextes tout en indiquant lequel de ces contextes est responsable de sa

[1]Un lien exprime une capacité d'adressage et non un droit d'accès dont la gestion est laissée aux politiques de gestion.

Concept	Notation	Domaine
traitement		$\{processus, noeud\}$
emplacement		$(adresse, noeud)$
copie		$(page, emplacement, version)$
lien		$(copie, contexte, typelien)$
typelien		$\{primaire, secondaire\}$
Espace logique	EL	$\{page\}$
Espace des traitements	ET	$\{processus\}$
espace physique	EP	$\{emplacements\}$
Espace des liens	$\mathcal{L}$	$\{lien\}$
Espace des copies	$\mathcal{C}$	$\{copie\}$
Père d'un contexte	$P(C)$	$contexte$
Fils d'un contexte	$F(C)$	$\{contexte\}$
Espace de gestion	$EG(C)$	$\{(page, traitement)\}$
Espace d'allocation	$EA(C)$	$\{copie\}$
Espace de visibilité	$EV(C)$	$\{copie\}$

TAB. 6.1 – Concepts du modèle de gestion mémoire hiérarchique.

gestion.

L'ensemble des copies liées à un contexte C est appelé *espace de visibilité* du contexte et noté $EV(C)$. L'ensemble des copies qui ont un lien primaire vers un contexte C est appelé espace d'allocation du contexte et noté $EA(C)$. Par définition de ces deux espaces nous avons l'inclusion $EA(C) \subseteq EV(C)$. La figure 6.4 donne un exemple de liens primaires et secondaires exprimant le partage physique de la copie $C2$ par les contextes $Ctx1.1$ et $Ctx1.2$.

Tout comme pour l'espace de gestion, l'espace d'allocation et de visibilité d'un contexte sont inclus dans ceux de son père, c.-à-d. $EA(C) \subseteq EA(P(C))$ et $EV(C) \subseteq EV(P(C))$. De plus, une copie ne peut être vue ou manipulée dans un contexte que si ce dernier est habilité à gérer les accès à la page que représente la copie. Cette contrainte s'exprime par l'inclusion $EA(C)|_{EL} \subseteq EV(C)|_{EL} \subseteq EG(C)|_{EL}$ où $E|_{EL}$ est la projection de l'espace E sur l'espace logique.

6.2.3 Dynamique du modèle

Le modèle de gestion est dynamique dans le sens où des instance des différents concepts présentés peuvent apparaître est disparaître à tout moment en fonction des besoins des serveurs, des accès effectués, etc.

Les contextes apparaissent et disparaissent lorsque ceux existants ne permettent

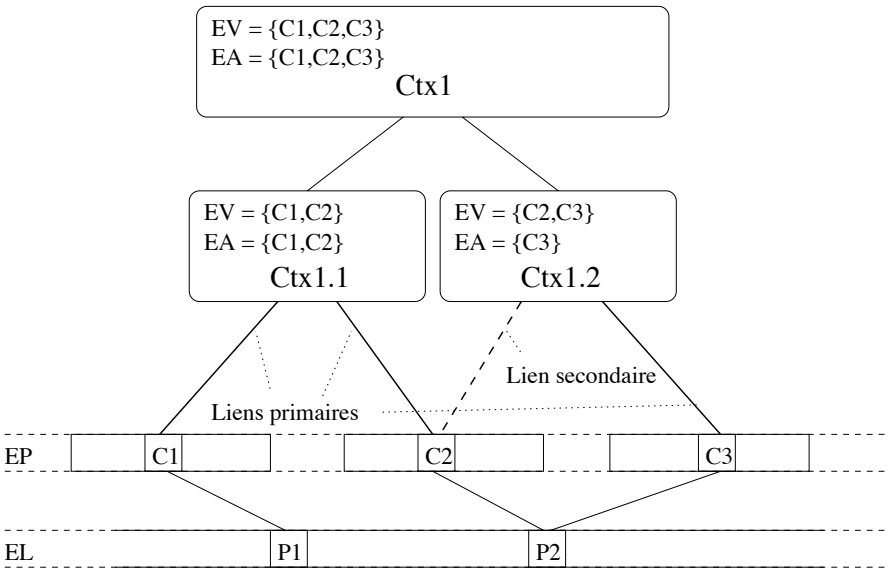


FIG. 6.4 – Exemple de liens primaires et secondaires.

pas une gestion adéquate de nouveaux types de comportements ou besoins d'accès. Ils peuvent également apparaître ou voir leur espace de gestion augmenter lorsqu'un accès détecté n'est inclus dans aucun des espaces de gestion des contextes terminaux.

Les copies apparaissent à mesure que de nouvelles pages sont accédées et disparaissent lorsque la taille de l'espace physique ne permet pas de toutes les conserver. Notons que l'augmentation de l'espace des copies peut être plus important lorsque les copies résidantes ne permettent pas de satisfaire des accès et qu'une réplication est nécessaire.

L'espace de visibilité d'un contexte augmente à mesure que sont détectés de nouveaux accès appartenant à son espace de contrôle. Cette augmentation se traduit par une augmentation des liens entre des copies et ses contextes terminaux. Il diminue à mesure que ces liens sont rompus lorsque les copies qui peuvent être accédées doivent être supprimées ou ne peuvent être conservées.

L'espace d'allocation d'un contexte augmente également à mesure que de nouveaux accès sont effectués dans son espace de contrôle. Toutefois, son augmentation est également dépendante de la nécessité de créer de nouvelles copies qui se traduit par une augmentation des liens primaires entre des copies et ses contextes terminaux.

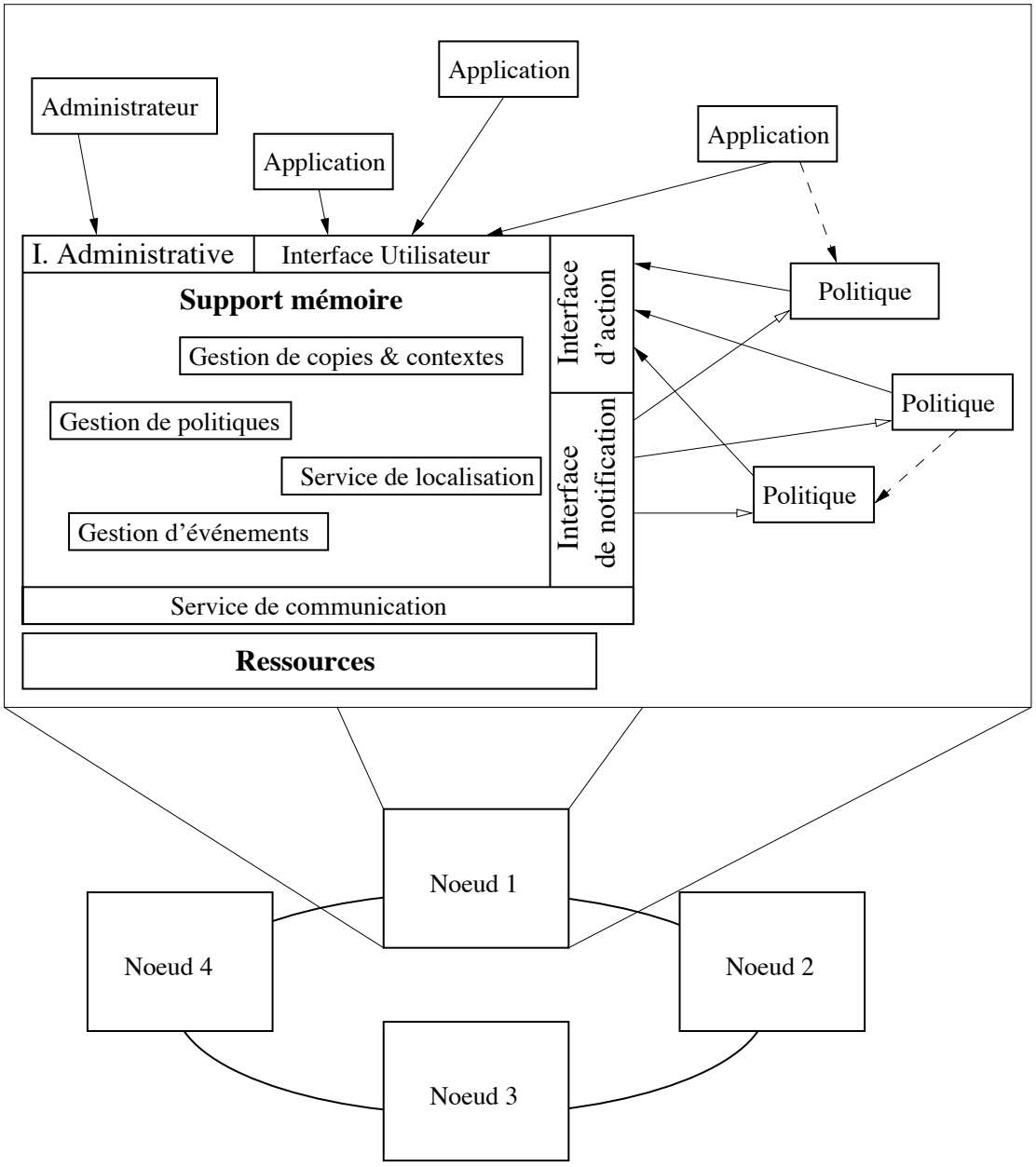


FIG. 6.5 – Architecture du support mémoire ADAMS.

6.3 Architecture

Le support mémoire que nous proposons met en oeuvre le modèle de gestion hiérarchique que nous venons de présenter. Comme le montre la figure 6.5, son architecture lui permet de servir d'intermédiaire entre les politiques et les ressources, entre les politiques et les applications et entre les politiques elles-mêmes. Il offre pour cela quatre interfaces :

- l'interface utilisateur correspondant à celle présentée dans le chapitre précédent permet essentiellement aux traitements de se déclarer et de déclarer leur accès,
- l'interface administrative permet entre autre d'ajouter et retirer des politiques de gestion,
- l'interface d'action permet aux politiques d'agir sur l'état du système,
- enfin, l'interface de notification permet aux politiques de réagir aux changements d'état du système.

Le support ne fait aucune hypothèse sur la manière dont sont définies les politiques et la façon dont elles interagissent et/ou coopèrent. Il s'assure que leurs actions respectent bien les contraintes imposées par le modèle et offre une architecture facilitant leur découplage.

L'approche que nous avons prise est d'adopter une architecture de type notification/réaction. Lorsque le support détecte un changement d'état du système, il génère des événements auxquels les politiques peuvent réagir à travers l'interface d'action. La réaction d'une politique peut être de mettre à jour son état interne de manière à prendre en compte les changements et/ou d'agir sur l'état du système via l'interface d'action. Cette deuxième possibilité est notamment importante lorsque le support détecte un état du système instable et exige des politiques de le remettre dans un état stable.

Les fonctions internes du support sont regroupées dans les services suivants :

- Le service de gestion de contextes & copies regroupe l'ensemble des mécanismes accessibles via l'interface d'action. Ils permettent notamment la manipulation de contextes et de copies et la création de liens entre les deux.
- Le service de localisation met en oeuvre des mécanismes et offre une interface aux politiques pour localiser l'ensemble des copies présentes en mémoire répartie.
- Le service de gestion des politiques permet d'ajouter et retirer des politiques, de les déclarer et de les rattacher aux contextes désirés.
- Le service de gestion d'événements permet la création de type d'événements, la signalisation d'événements et leur routage vers les politiques.

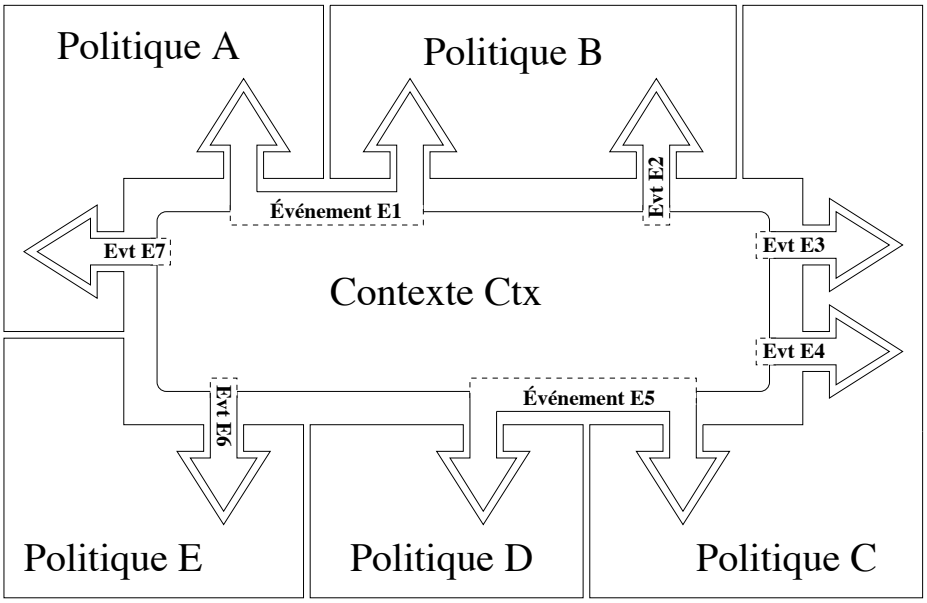


FIG. 6.6 – Rattachement de politiques à un contexte.

Les sous-sections suivantes présentent plus en détails ces différents services.

6.3.1 Gestion de politiques

Nous considérons qu'une politique est un programme compilé et conçu en connaissance du support. Afin d'y être intégrée, elle se doit d'exposer une interface qui permettra au support de lui communiquer des événements auxquels elle pourra réagir. Elle est constituée d'un ensemble de procédures de traitement supposées prendre des décisions et effectuer des actions répondant à des types d'événements particuliers.

La figure 6.6 illustre comment les politiques sont intégrées au support. Nous avons vu que le modèle a pour but de permettre une gestion hiérarchique de la mémoire où à chaque contexte peut être attaché un ensemble de politiques de gestion. Ainsi, le service de gestion de politiques permet de rattacher une politique à un contexte en spécifiant les types d'événements auxquels elle désire réagir et en fournissant pour chaque type d'événement l'adresse d'une procédure de traitement. Dans l'exemple, les politiques A à E sont rattachées au contexte Ctx. Les politiques A et B sont toutes deux abonnées à l'événement $E1$ mais réagissent également chacune à un événement particulier ($E7$ et $E2$). La politique C réagit à trois types d'événements dont un (événement $E5$) est partagé avec la politique D. La politique E quant à elle réagit uniquement à l'événement $E6$ auquel elle est la seule à réagir.

Ajouter une politique au support se déroule comme suit. La politique est compilée,

puis un service de chargement permet de lier dynamiquement le code de la politique à celui du support. Une fois le chargement effectué, la politique peut utiliser les services fournis par de simples appels de procédure. Elle doit s'enregistrer auprès du support de manière à obtenir un identifiant de politique qu'elle utilise pour s'abonner aux types d'événements auxquels elle désire réagir. Tant qu'elle demeure chargée, elle peut s'attacher ou se détacher de n'importe quel contexte existant. Tant qu'elle est attachée à un contexte, tous les événements concernant le contexte et auxquels elle est abonnée lui sont signalés. Notons que l'attachement ou le détachement d'une politique à un contexte peut cependant être effectué par n'importe quelle autre politique pourvue qu'elle possède son identifiant.

Une même politique peut être rattachée simultanément à plusieurs contextes rendant ainsi possible le partage de son code pour la gestion de différents contextes. La politique est cependant responsable de gérer ce partage en maintenant si nécessaire différents états de gestion pour les différents contextes.

Le rattachement d'une politique à un contexte ne peut être effectué que si le contexte est désactivé (ce qui est le cas juste près la création de ce dernier). Lors de l'activation du contexte, un événement est signalé afin de permettre aux politiques d'initialiser leur état de gestion. Changer la manière dont est géré un contexte consiste à détacher une ou plusieurs politiques du contexte et d'en attacher de nouvelles. Il est donc nécessaire pour cela de désactiver temporairement le contexte. Cela est également signalé implicitement aux politiques concernées de sorte qu'elles peuvent se terminer proprement et détruire leur état interne. Le service prévoit un zone d'échange non structurée pour chaque politique afin de permettre à une politique remplacée de laisser des indications à une politique remplaçante. En l'absence d'indications, cette dernière peut toutefois interroger le support afin de déterminer l'état du contexte auquel elle est rattachée et de construire son état de gestion en conséquence.

Le support n'ayant aucune information sur la sémantique des politiques, il n'impose aucune contrainte concernant leur abonnement à des événements et leur rattachement à des contextes. Cependant le support pré-défini des types d'événements correspondant à des changements d'état bien particuliers et signalés d'une manière bien définie. Pour cette raison, nous préconisons de définir, selon le type de contexte, les politiques suivantes :

- au niveau d'un contexte intermédiaire :
  - une politique d'organisation de la gestion du contexte responsable de la création et destruction de sous-contextes et de l'attachement des politiques de gestion à ses sous-contextes,
  - une politique de cohérence pour la gestion des pages répliquées dans le contexte,

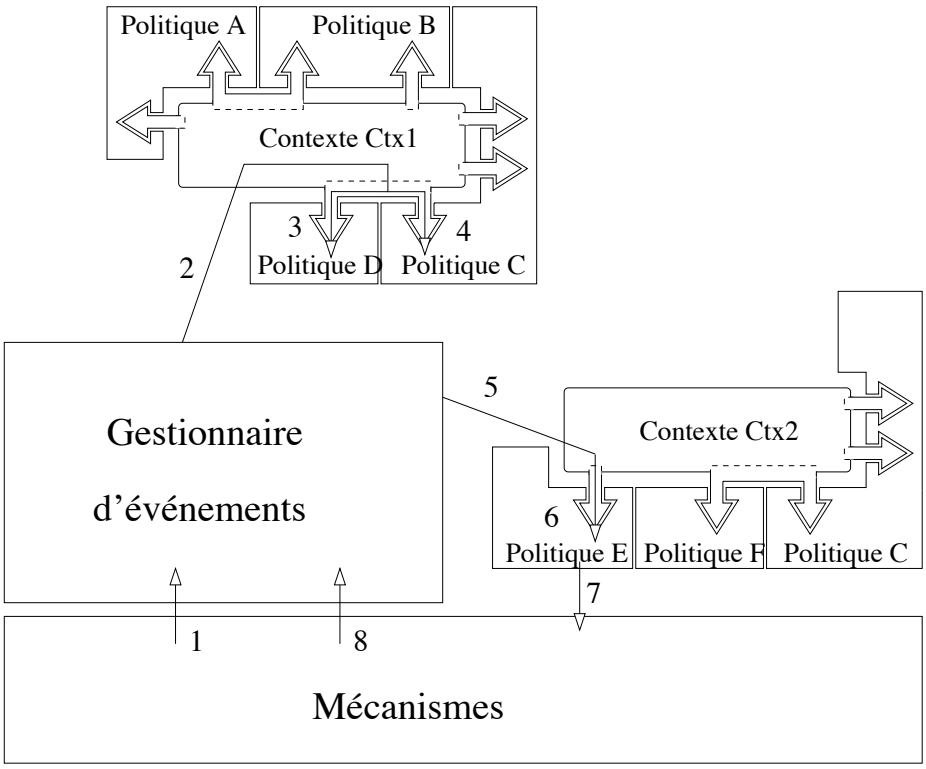


FIG. 6.7 – Signalisation des événements.

- une politique de synchronisation pour les pages partagées dans le contexte,
- une politique d'allocation responsable de l'allocation d'emplacements entre les différents sous-contextes,

– au niveau d'un contexte terminal :
- une politique de remplacement,
- une politique de préchargement

Les identifiants de politiques sont uniques et répartis. Cela signifie qu'une même politique peut être rattachée à un contexte réparti et/ou à différents contextes sur différentes machines. Le code de la politique doit cependant être préalablement chargé sur chaque machine où l'événement peut avoir lieu. L'enregistrement d'une politique, son abonnement à des événements et sont rattachement à des contextes ne doivent être effectués qu'une seule fois. Lors du rattachement, le support détecte automatiquement les machines concernées par le contexte afin de mettre à jour ses structures de contrôle réparties.

politique A	en premier	en dernier	avant(B)	après(B)	sans importance
politique B					
en premier	F	V	F	V	V
en dernier	V	F	V	F	V
avant(A)	F	V	F	V	V
après(A)	V	F	V	F	V
sans importance	V	V	V	V	V

TAB. 6.2 – Matrice de compatibilité des priorités de signalisation des politiques.

6.3.2 Gestion d'événements

Comme le montre la figure 6.7, le gestionnaire d'événement (GE) sert d'intermédiaire entre les politiques et le support. Il offre tout d'abord une fonction permettant de définir des types d'événements. Elle est appelée de façon interne lors de l'initialisation du support afin de créer les types d'événements propres au support. Les politiques peuvent s'abonner ensuite à ces événements comme expliqué précédemment.

Lorsque le support détecte un changement d'état du système, il demande au GE de signaler l'événement correspondant en précisant le contexte concerné. Le GE permet de plus de passer des arguments dans une zone mémoire non structurée. Il recherche ensuite les politiques rattachées au contexte et signale à chacune l'occurrence de l'événement. Il appelle pour cela la procédure de traitement indiquée par la politique lors de son abonnement à l'événement en passant comme argument l'identifiant et le type de l'événement, l'identifiant du contexte et un pointeur vers la zone mémoire contenant les arguments.

L'ordre dans lequel sont appelées les politiques peut être imposé de la manière suivante. Lors d'un abonnement, une politique précise si elle doit être appelée en première, en dernière, après ou avant une autre politique déjà abonnée, ou bien si cela lui est égal. Lors de son rattachement à un contexte, le support vérifie que ses desiderata ne sont pas en contradiction avec ceux des politiques déjà enregistrées, conformément à la matrice de compatibilité présentée dans le tableau 6.2.

Les événements sont notifiés immédiatement aux politiques en appelant successivement et de manière synchrone les différentes procédures de traitement. Un événement est propagé de politique en politique et ne peut par conséquent être traité que par une seule politique à la fois. Toutefois, différents événements peuvent être traités simultanément par différentes politiques ou par une même politique.

Le support permet à une politique de bloquer temporairement la propagation d'un événement. Elle peut également décider de stopper définitivement sa propagation afin

qu'il ne soit pas signalé aux autres politiques. De ce fait, la possibilité de préciser l'ordre dans lesquelles les politiques sont signalées gagne de l'importance du fait qu'il est possible de "cacher" l'occurrence d'un événement.

La fonction de déclaration de type d'événement prend comme paramètre une procédure de pré-traitement et une de post-traitement appelées pour chaque contexte respectivement avant et après le signalement de l'événement aux politiques. La procédure de post-traitement peut mettre en oeuvre plusieurs fonctions. Elle peut par exemple servir à router l'événement de contextes en contextes ou effectuer des vérifications de fin de traitement.

Certains types d'événements définis par le support correspondent à la détection d'une instabilité de l'état du système. Cette procédure peut ainsi être utilisée afin de vérifier que les politiques qui sont sensées s'être abonnées à l'événement sont parvenues à stabiliser cet état. Dans le cas contraire, le support effectue des actions de manière à stabiliser lui-même l'état et pouvant éventuellement conduire à la destruction du contexte et des politiques rattachées estimant que celles-ci ont un comportement inacceptable. De la sorte, le support prévient les politiques de conduire le système à un état non recouvrable. Par exemple, lors de la signalisation d'un défaut de page, le système s'assure qu'une copie de la page est effectivement chargée en mémoire par une des politiques rattachées au contexte.

Séparer la gestion du GE du reste du support permet d'exposer les fonctions de création de type d'événement et de signalisation directement aux politiques. Il est donc possible de définir de nouveaux types d'événement, et de les signaler aux contextes désirés. De cette manière, le support reste suffisamment ouvert pour ne pas figer complètement la définition de l'état du système et permettre son extension.

6.3.3 Localisation des copies

Certaines politiques nécessitent de localiser les copies présentes dans l'espace physique. Un service de localisation maintient pour cela une table virtuelle en interne dont un exemple d'instantiation est présenté dans le tableau 6.3. Il offre la fonction `localiser_copies(page, version, noeud, contexte, type_de_lien)` permettant d'interroger cette table, chaque argument permettant de restreindre la requête. Cette fonction permet par exemple de localiser :

- l'ensemble des copies du système (`localiser_copies(all, all, all, all, all)`),
- l'ensemble des copies de la page $p2$ sur le noeud $n2$ (`localiser_copies(p2, all, n2, all, all)`),

copie	page	version	noeud	contexte terminal	type de lien
c1	p1	v3	n2	ctx2	primaire
c2	p1	v1	n2	ctx1	primaire
c3	p1	v1	n2	ctx3	secondaire
c4	p2	v0	n1	ctx1	primaire
c5	p2	v0	n2	ctx1	primaire
c6	p2	v0	n2	ctx3	secondaire

TAB. 6.3 – Table de localisation des copies.

- l'ensemble des copies dans l'espace de visibilité du contexte $ctx2$ (`localiser_copies(all, all, all, ctx2, all)`)
- l'ensemble des copies dans l'espace d'allocation du contexte $ctx2$ (`localiser_copies(all, all, all, ctx2, lien_primaire)`),
- l'ensemble des copies jumelles de la copie $c1$ (`localiser_copies(c1.page, c1.version, all, all, all)`)

Nous présentons dans le chapitre suivant une implantation de ce service permettant de minimiser le coût de localisation.

6.3.4 Gestion des communications

Certains contextes nécessitent la mise en oeuvre de politiques de gestion réparties dont une instance est chargée sur chaque machine. Ces politiques nécessitent généralement de pouvoir faire communiquer leurs différentes instances. Un service de communication permet d'échanger des messages et prend comme argument l'identifiant de la politique à contacter, le numéro du noeud de l'instance et une zone mémoire contenant le message. Bien que le gestionnaire de communication et de localisation puisse être utilisés de manière indépendante, leur intégration est préférable afin de factoriser les coûts de localisation et de communication. Le service permet notamment de préciser le noeud de manière logique à l'aide des identifiants de page, de version et de contexte. Une implantation de ce service est également présentée plus en détail dans le chapitre suivant.

6.4 Interfaces

Cette section spécifie les interfaces d'action et de notification de notre support et dont les principales fonctions et principaux événements sont présentés dans les

fonction	cohérence	synchronisation	allocation	remplacement	préchargement	organisation
initier_accès					X	
créer_contexte						X
activer_contexte						X
désactiver_contexte						X
détruire_contexte						X
allouer_emplacement			X			
libérer_emplacement			X			
réclamer_emplacement			X			
localiser_copies	X			X	X	
créer_copie	X			X		
créer_lien	X			X		
changer_version	X					
changer_destructibilité	X					
recopier_valeur	X					
lire_page	X					
détruire_lien	X			X		
bloquer_processus		X				
réveiller_processus		X				
préparer_accès					X	

TAB. 6.4 – Interface d'action du support ADAMS.

tableaux 6.4 et 6.5. Le tableau 6.4 donne pour chaque fonction les politiques utilisatrices attendues. Le tableau 6.5 quant à lui donne pour chaque type d'événement, les politiques sensées être concernées. Il indique en outre si la signalisation de l'événement correspond ou non à la détection d'une instabilité de l'état du système et une indication sur le type de parcours de l'événement. Un événement descendant est signalé en descendant l'arbre des contextes de la racine vers les feuilles tandis qu'un événement ascendant est remonté des feuilles vers la racine. Un événement local est normalement signalé au contexte concerné.

La suite de cette section montre comment ces types d'événement et fonction offre un support pour une gestion hiérarchique d'une mémoire répartie.

événement	parcours	état	cohérence	synchronisation	allocation	remplacement	préchargement	organisation
`evt_début_accès`	descendant	stable		X		X		
`evt_fin_accès`	ascendant	stable		X		X		
`evt_prépare_accès`	ascendant	stable	X	X	X			
`evt_début_traitement`	descendant	stable		X				
`evt_fin_traitement`	ascendant	stable		X				
`evt_pas_de_ctx_terminal`	descendant	instable						X
`evt_pas_de_lien`	local	instable	X					
`evt_lien_créé`	descendant	stable	X					
`evt_lien_détruit`	ascendant	stable	X					
`evt_contexte_activé`	local	stable	X		X	X	X	X
`evt_contexte_désactivé`	local	stable	X		X	X	X	X
`evt_contexte_détruit`	local	stable	X		X	X	X	X
`evt_emplacement_alloué`	local	stable			X		X	
`evt_emplacement_libéré`	local	stable			X		X	
`evt_réclame_emplacement`	ascendant	instable			X	X		
`evt_fin_écriture`	ascendant	stable	X				X	
`evt_change_destructibilité`	local	stable				X		
`evt_change_réplication`	local	stable				X		

TAB. 6.5 – Interface de notification du support ADAMS.

6.4.1 Support pour l'organisation

L'organisation de l'arbre de contextes est laissée libre à l'utilisateur du support. Cette organisation peut être hiérarchisée en rattachant à un contexte une politique d'organisation des sous-contextes correspondant. Cette politique est responsable de la création et destruction des contextes, de faire évoluer leurs espaces de gestion et de leur associer des politiques de gestion adéquates.

La création et la destruction d'un contexte peuvent être effectuées en réaction aux événements `evt_début_traitement` et `evt_fin_traitement`. Toutefois, la nécessité de créer un contexte peut être signalée par l'événement `evt_pas_de_ctx_terminal`. En effet, à chaque accès détecté, le support doit retrouver l'ensemble de contextes concernés et notamment le contexte terminal auquel un lien vers une copie de la page accédée doit être créé. Il s'appuie pour cela sur la valeur des espaces de gestion des contextes et de leur inclusion. Lorsqu'aucun contexte terminal n'est trouvé, l'événement `evt_pas_de_ctx_terminal` est propagé de manière ascendante en commençant par le plus petit contexte intermédiaire dont l'espace de gestion inclut l'accès. Une politique d'organisation est sensée le récupérer afin de créer le ou les contextes correspondants ou bien d'augmenter l'espace de gestion de contextes existants. Cet événement est instable du fait que tous les accès doivent appartenir à l'espace de visibilité d'au moins un contexte terminal.

6.4.2 Support pour la cohérence et la synchronisation

Le support permet de définir différentes politiques de cohérence et de synchronisation pour différents contextes. Bien qu'une gestion hiérarchique de la cohérence de pages répliquées est possible, certaines politiques de cohérence nécessitent de contrôler complètement la réplication à leur niveau. Lors de la première création d'une copie d'une page, le support permet à la politique créatrice d'indiquer qu'elle désire garder le contrôle de la création et destruction d'autres copies de la même page. Dans ce cas, la politique est considérée comme étant *la* politique de cohérence et de réplication de la page, tout autre demande de création ou destruction d'une copie de la page dans un contexte différent étant refusée.

Le support rend possible une gestion hiérarchique de la synchronisation des accès aux pages : la synchronisation de deux ensembles d'accès est effectuée au niveau d'un contexte et la synchronisation entre les accès d'un ensemble est effectuée dans un contexte fils. Le support rend donc possible la gestion hiérarchique de la C&S la plus plausible où la cohérence est gérée dans un contexte proche de la racine et la synchronisation effectuée au niveau de ce contexte et éventuellement au niveau des

contextes de niveau inférieur.

La politique de cohérence différencie si nécessaire les différentes copies d'une page répliquée à l'aide des versions. Rappelons que deux copies qui ne peuvent pas être échangées sont supposées avoir des numéros de version différents. Comme nous le verrons par la suite, cela est important pour permettre la gestion d'une politique de remplacement répartie. Indiquer simplement à cette politique si une page est répliquée n'est pas suffisant pour lui permettre d'évaluer son utilité étant donné que les copies peuvent avoir des valeurs différentes. Comme seule la politique de cohérence de la page est capable de distinguer les différentes copies, le support pour les versions permet une meilleure intégration des deux types de politique.

Cinq types d'événements définis par le support servent aux politiques de C&S :

- `evt_début_traitement` et `evt_fin_traitement` sont les événements correspondant aux fonctions `début_traitement` et `fin_traitement` de l'interface utilisateur. Réagir à ces événements permet de mettre par exemple en oeuvre des propriétés d'isolation.
- `evt_début_accès` et `evt_fin_accès` correspondent aux fonctions `début_accès` et `fin_accès` de l'interface utilisateur. Un argument dont le type est extensible permet de préciser s'il s'agit d'un lecture, d'une écriture ou d'un autre type d'accès.
- `evt_pas_de_lien` est signalé par le support suite à un accès lorsqu'il n'existe aucun lien entre une copie de la page accédée et le contexte terminal gérant cet accès. Cet événement correspond à un état instable qui doit être stabilisé par la création d'un lien vers la copie.

Les événements `evt_début_traitement`, `evt_début_accès`, `fin_traitement` et `evt_fin_accès` sont signalés à tous les contextes inférieurs au contexte dont l'accès est dans l'espace de gestion, les deux premiers étant descendants et les deux derniers ascendants. Ces événements n'ont pas de procédures de pré ou post-traitement associées.

L'événement `evt_pas_de_lien` est signalé uniquement à la politique de cohérence de la page. Elle peut décider de lier le contexte à une nouvelle copie ou bien de le lier à une copie déjà existante localement. La fonction de localisation permet à la politique de retrouver l'ensemble des copies résidantes en mémoire locale. Lors de la création d'un lien avec une nouvelle copie, le support exige que le lien créé soit primaire. Si cela est le cas, la fonction réclame un emplacement au contexte terminal afin qu'il prenne en compte la copie dans son espace d'allocation. La procédure de post-traitement associée à l'événement vérifie que la politique de cohérence est effectivement parvenue à lier la page au contexte terminal.

Lors de la création d'une nouvelle copie, la politique peut décider de l'initiali-

ser à partir d'une copie déjà existante soit localement soit en mémoire distante. Ici encore, la fonction de localisation permet de retrouver de telles copies. La fonction `recopier_valeur` permet ensuite à la politique de recopier la valeur d'une copie vers une autre. Si la valeur ne peut être obtenue qu'à partir du disque, la fonction `lire_page` permet de lire l'image permanente de la page et de l'écrire dans l'emplacement de la copie.

L'attribution ou le changement d'un numéro de version peuvent être effectués à tout moment à l'aide de la fonction `changer_version` qui retourne un identifiant unique. Cet identifiant peut ensuite être utilisé afin de retrouver l'ensemble des copies d'une même version à l'aide de la fonction `localiser_copie`.

Lorsqu'une copie n'est plus jugée utile par les politiques de remplacement de contextes terminaux auxquels elle est liée, elle doit être détruite par le support. Avant d'effectivement la détruire, le support signale l'événement `evt_copie_détruite` à la politique de C&S. De cette manière, les modifications apportées à la copie peuvent être répercutées sur disque. Il peut cependant arriver qu'une copie ne puisse être détruite car elle contient des modifications qui ne peuvent être reportées immédiatement dans l'espace de stockage[2]. Le support fournit pour cela un espace disque permettant de stocker de telles copies. Il peut cependant être intéressant de préserver ces copies en priorité. Afin de faciliter la coopération entre les politiques de cohérence et de remplacement, le support associe à chaque page un indicateur de destructibilité. Une fonction permet de changer cet indicateur, générant un événement correspondant au niveau du contexte terminal et pouvant être récupéré par la politique de remplacement (cf. sous-section 6.4.4).

6.4.3 Support pour l'allocation

Le support pour l'allocation que nous offrons se base sur l'étude menée dans la section 4.2. L'objectif est de permettre tout type d'allocation tout en automatisant et minimisant le plus possible l'interaction entre les différents contextes. Le modèle sur lequel nous nous appuyons permet de hiérarchiser la gestion de l'allocation. Un contexte décide la manière dont les emplacements dont il dispose sont répartis parmi ses sous-contextes qui à leur tour peuvent décider comment les répartir parmi leur sous-contextes respectifs. Remarquons que grâce au concept d'espace de contrôle, le modèle permet tous les modes de partage de la mémoire physique. Il est en effet possible de diviser un contexte en sous-contextes soit par les données soit par les

[2]Cela arrive notamment lorsqu'il s'agit d'une page modifiée par une transaction non validée et que le serveur gère un journal contenant "l'image après" des données.

traitements soit par les deux.

À chaque contexte ctx sont associées quatre valeurs d'allocation dont les deux dernières sont fixées une fois pour toutes lors de la création du contexte :

- $ctx.n$ représente le nombre d'emplacements qui ont été alloués à ctx par son père,
- $ctx.k$ représente le nombre d'emplacements que le contexte a alloué à ses fils s'il s'agit d'un contexte intermédiaire (dans ce cas, $ctx.k = \sum_{ctx_i \in F(ctx)} ctx_i.n$), et le nombre de copies pour lesquelles le contexte possède un lien primaire s'il s'agit d'un contexte terminal,
- $ctx.min$ fixe l'allocation minimale garantie par le contexte père,
- $ctx.max$ fixe l'allocation maximale que son père lui laisse espérer.

Ces valeurs d'allocation sont soumises aux contraintes suivantes :

- $ctx.n \geq ctx.k$, c.-à-d. un contexte intermédiaire ne peut allouer à ses fils plus d'emplacements que son propre père ne lui a alloués, et un contexte terminal ne peut posséder plus de liens primaires que le nombre d'emplacements que son père ne lui a alloués ,
- $ctx.min \geq \sum_{ctx_i \in F(ctx)} ctx_i.min$, c.-à-d. un contexte ne peut garantir à ses fils plus d'emplacements que ce que son propre père ne lui garantit,
- $\forall ctx_i \in fils(ctx), ctx_i.max \leq ctx.max$, c.-à-d. un contexte ne peut laisser espérer à chacun de ses fils plus que son père ne lui laisse espérer.

L'allocation d'un emplacement à un contexte peut être soit effectuée explicitement par son père, soit réclamé par le contexte. De même la libération d'un emplacement peut être soit effectuée explicitement par un contexte soit réclamée par son père. Dans les deux cas, le support assure que les contraintes d'allocation sont respectées et met à jour automatiquement les valeurs d'allocation des contextes concernés.

Lors de l'établissement d'un lien primaire entre une copie et un contexte terminal, un emplacement est implicitement réclamé à ce dernier. En fait, tout se passe comme si une copie était un contexte dont les valeurs minimale et maximale d'allocation sont égales à 1. Par conséquent l'emplacement demandé ne peut être refusé par le contexte.

Les fonctions `réclame_emp`, `alloue_emp`, et `libère_emp` sont décrites par l'algorithme 1. La dernière s'appuie sur l'événement `evt_réclame_emp(demandeur,force)` et dont les procédures de pré et post-traitement sont décrites par l'algorithme 2. L'argument `demandeur` donne l'identifiant du demandeur qui peut être soit le père du contexte auquel est signalé l'événement soit un de ses fils. L'argument `force` indique s'il s'agit d'une demande forte (c.-à-d. un ordre) qui *doit* être satisfaite ou une demande faible qui *peut* être satisfaite. Une demande est forte si elle va d'un fils à son père et que la valeur d'allocation minimale du premier n'est pas atteinte. Elle

Algorithm 1 Fonctions `réclame_emp`, `alloue_emp` et `libère_emp`.

```
réclame_emp(de_ctx,a_ctx) {
  si ((a_ctx = de_ctx.père) et (de_ctx.n < de_ctx.min)) ou
     ((de_ctx = a_ctx.père) et (a_ctx.n > a_ctx.min))
    signale(a_ctx, evt_réclame_emp(de_ctx, forte)
  sinon
    signale(a_ctx, evt_réclame_emp(de_ctx, faible)
}

alloue_emp(de_ctx, a_ctx) {
  de_ctx.k = de_ctx.k + 1
  a_ctx.n = a_ctx.n + 1
}

libère_emp(de_ctx, a_ctx) {
  de_ctx.n = de_ctx.n - 1
  a_ctx.k = a_ctx.k - 1
}
```

l'est également si elle va d'un père à un fils et que la valeur minimale du dernier est dépassée. Dans les cas contraires, la demande est faible indiquant qu'il s'agit d'une tentative de réclamation qui peut être refusée par la politique.

La procédure de pré-traitement vérifie si l'allocation peut être effectuée sans intervention de la politique, c'est à dire si $ctx.k < ctx.n$. Si c'est le cas, elle invoque implicitement la fonction `alloue_emp` et termine l'événement. Si ce n'est pas le cas et que l'allocation maximale n'est pas atteinte, elle réclame implicitement un emplacement au contexte père, la force de la demande dépendant du dépassement ou non de la valeur d'allocation minimale. Dans le cas contraire ou si la demande n'a pu être satisfaite, l'événement est signalé aux politiques du contexte local. S'il s'agit d'un contexte terminal, une politique de remplacement est sensée récupérer l'événement afin de libérer un emplacement en détruisant un des liens primaires. S'il s'agit d'un contexte intermédiaire, une politique d'allocation est sensée récupérer l'événement afin d'allouer un emplacement supplémentaire au contexte demandeur et peut être amenée pour cela à réclamer des emplacements à ses autres fils. La procédure de post-traitement vérifie simplement que les demandes fortes sont effectivement satisfaites.

Notons que lorsqu'un emplacement est demandé à un contexte, celui peut éventuellement libérer ou allouer plusieurs emplacements. Les demandes suivantes pourront ainsi être satisfaites sans avoir besoin de contacter à nouveau le contexte, réduisant

Algorithm 2 Pré- et post-traitements de l'événement `evt_réclame_emp`.

```
evt_réclame_emp.pré(de_ctx, a_ctx, force) {
  // demande du fils au père
  si (a_ctx = de_ctx.père) {
    si (force = forte)
      si (a_ctx.n > a_ctx.k) {
        alloue_emp(a_ctx, de_ctx)
        retourne evt_terminé }
      sinon-si (a_ctx.n < a_ctx.min)
        réclame_emp(a_ctx, a_ctx.père, forte)
      sinon-si (a_ctx.n < a_ctx.max)
        réclame_emp(a_ctx, a_ctx.père, forte)
      sinon-si (a_ctx.n = a_ctx.max)
        réclame_emp(a_ctx, a_ctx.père, faible)
        si (a_ctx.n > a_ctx.k)
          retourne evt_termine
    retourne evt_continue
  }
  // demande du père au fils
  sinon-si (de_ctx = a_ctx.père) {
    si (force = forte) et (a_ctx.n < a.ctx.k) {
      libère_emp(a_ctx, de_ctx)
      retourne evt_terminé }
    retourne evt_continue
  }
}

evt_réclame_emp.post(de_ctx, a_ctx, force) {
  si (force = forte) et (de_ctx.n = de_ctx.k)
      détruire le contexte fautif
}
```

ainsi le nombre d'interactions.

6.4.4 Support pour le remplacement et le préchargement

Le support permet de définir différentes politiques de remplacement pour différents contextes. Chaque contexte peut donc décider de la manière d'évaluer l'utilité des copies qui sont dans son espace d'allocation. Elle dispose pour cela :

- des événements `evt_début_accès` et `evt_fin_accès` qu'elle peut récupérer afin d'être avertie lorsqu'une copie est accédée et mettre à jour ses statistiques d'accès,
- de l'indicateur de destructibilité mis à jour par la politique de C&S et dont la modification génère l'événement `evt_change_destructibilité` qui peut être récupéré par la politique,
- de la fonction de localisation qui permet de déterminer si une copie est répliquée.

Notons que le service de localisation permet de prendre en compte la réplication d'une page quand bien même ses copies appartiennent à l'espace d'allocation de différents contextes. De plus, la possibilité d'attribuer des versions aux copies est pour les politiques un moyen de ne s'assurer que deux copies sont interchangeables. L'événement `evt_change_réplication` est signalé au contexte terminal à chaque fois qu'une copie devient répliquée ou unique.

Lorsqu'un lien primaire est créé par une politique de C&S entre une copie et un contexte, un emplacement est réclamé au contexte terminal afin d'ajouter la copie à l'espace d'allocation de ce dernier. Comme nous l'avons vu dans la sous-section 6.4.3, lorsque plus aucun emplacement n'est disponible et que le père du contexte refuse de lui en allouer un supplémentaire, l'événement `evt_réclame_emp` est signalé à la politique de remplacement. Cette dernière doit alors satisfaire la demande, c'est-à-dire choisir une des copies dans l'espace d'allocation du contexte et changer un des liens primaires en lien secondaire.

Un des avantages de notre modèle est qu'il permet de considérer l'utilité d'une page dans différents contextes. En fait, l'événement `evt_lien_créé` est signalé également lorsqu'un lien secondaire est créé et la copie liée est ajouté à l'espace de visibilité du contexte. De cette manière la politique de remplacement peut prendre en compte l'ensemble des copies de l'espace de visibilité et non se restreindre à l'espace d'allocation, les événements `evt_début_accès`, `evt_fin_accès` et `evt_change_réplication` étant signalés pour toutes les copies de l'espace de visibilité du contexte.

Lorsqu'un lien primaire est détruit, l'événement `evt_lien_détruit` est signalé à chaque contexte qui possède un lien secondaire sur la copie. Comme l'indique la

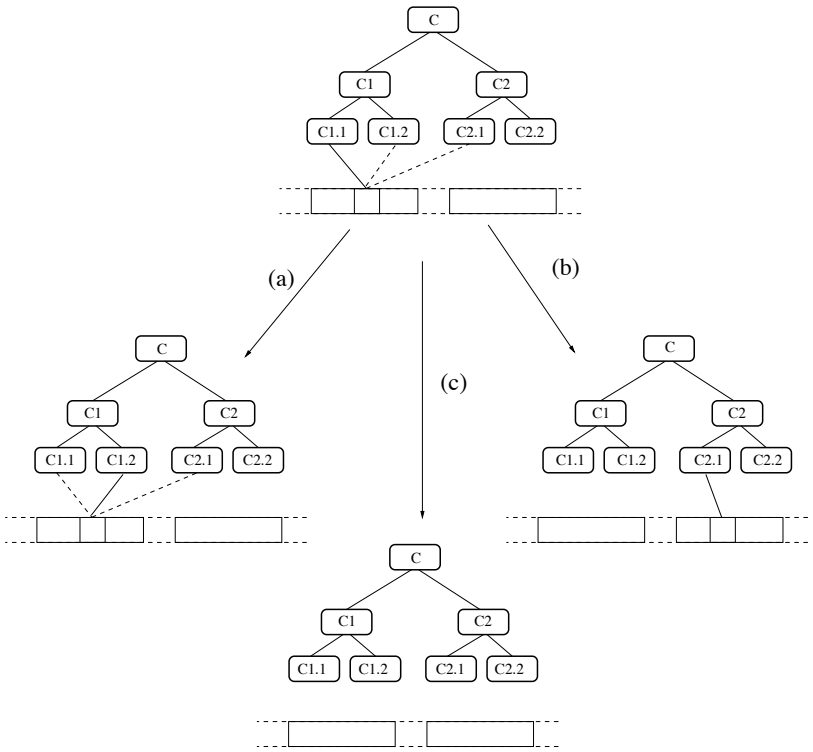


FIG. 6.8 – Remplacement d'une copie partagée par plusieurs contextes.

figure 6.8, chacune des politiques de remplacement associées peut alors choisir selon l'estimation qu'elle a de la copie :

- (a) de changer le lien secondaire en lien primaire et faire rentrer la page dans son espace d'allocation, détruisant éventuellement le lien primaire d'une autre copie,
- (b) de préserver sa valeur en créant une autre copie en mémoire distante,
- (c) de ne rien faire.

Si aucune politique ne choisit la première solution, la copie est détruite, les liens secondaires sont détruits et la destruction est signalée à la politique de C&S (cf. sous-section 6.4.2).

Le support pour le préchargement est relativement simple puisqu'il consiste simplement à fournir la fonction `préparer_accès` et l'événement `evt_prépare_accès` associé. Cet événement est sensé être récupéré par une politique de C&S au même titre que l'événement `evt_début_accès`. De cette manière, la politique de préchargement indique quelles pages risquent d'être accédées et la politique de C&S décide comment anticiper cet accès (création d'une copie locale et/ou d'un lien, récupération de verrous, etc.). Les politiques d'allocation peuvent également être intéressées par cet événement si elles jugent nécessaire d'anticiper une augmentation de la taille de

l'espace d'allocation.

6.5 Conclusion

Dans ce chapitre, nous avons présenté ADAMS, un support mémoire adaptable pour serveurs de données répartis. ADAMS s'appuie sur un modèle de gestion hiérarchique basé sur la notion de contexte auquel sont rattachés des espaces de gestion, de visibilité et d'allocation. L'espace de gestion permet de définir proprement les responsabilités de gestion des différents contextes tandis que les espaces de visibilité et d'allocation offrent une représentation de la manière dont les copies sont liées aux contextes.

Afin de faciliter l'interaction entre les politiques et le support et entre les politiques elles-mêmes, ADAMS met en oeuvre des mécanismes pour la gestion d'événements (création, signalisation, abonnement). Le support utilise ces mécanismes afin d'offrir des interfaces d'action et de notification. L'interface de notification permet aux politiques de réagir à des changements d'état du système tandis que l'interface d'action leur permet de modifier cet état.

En s'intercalant entre les politiques et les ressources, le support reste maître de l'intégrité du système. Le mode de communication choisi permet de bien séparer les différentes politiques tout en permettant leur intégration. Les concepts de liens et de version définis dans le modèle facilitent la cohabitation des politiques de remplacements entre elles et avec les politiques de C&S. Une spécification des interfaces d'action et de notification a été proposée et nous avons finalement montré comment ces interfaces peuvent être utilisées par les différents types de politique. Nous avons par conséquent montré l'apport de notre support pour une gestion adaptable de la mémoire répartie de serveurs de données.

En conclusion, notre support offre une infrastructure pour la construction de supports dont la gestion mémoire est adaptable, dont le grain d'adaptabilité est fin et pouvant faire cohabiter différentes politiques pour différents besoins et comportements.

Chapitre 7

Implantation et exploitation du support mémoire ADAMS

7.1 Introduction

Dans le chapitre précédent nous avons spécifié le support mémoire adaptable ADAMS facilitant l'implantation de gestionnaires mémoires de serveurs de données. L'objectif de ce chapitre est de montrer comment nous avons pu appliquer cette proposition à un contexte particulier. Le contexte choisi est celui d'une mémoire virtuelle répartie partagée (MVRP) et plus particulièrement le système Arias. Ce système constitue un support pour la mise en oeuvre d'applications réparties en offrant l'abstraction d'une MVRP et un ensemble de mécanismes permettant une gestion adaptable des ressources d'une grappe de machines.

Après avoir présenté l'architecture et les principales caractéristiques du système Arias, nous montrons comment nous étendons et utilisons ce dernier pour implanter notre proposition. Nous indiquons les contraintes imposées par le contexte des MVRP et par le système lui-même avant de détailler les structures de données gérées par le système et les algorithmes des principales fonctions. Enfin, nous montrons comment ADAMS peut être utilisé afin d'implanter un serveur de données, et nous montrons comment il est possible de changer de politique en cours d'exécution.

7.2 Contexte d'implantation

Le système Arias a été développé dans un contexte semi-industriel entre l'INRIA (projet SIRAC), la société Bull (action Mescaline du GIE Dyade) et le LSR (projet STORM). Commencé en 1995, il a abouti à la réalisation de deux prototypes. Notre

travail s'articule autour du deuxième prototype dont l'implantation a été, pour des raisons politiques, interrompue courant 1998.

Arias offre un support pour la conception d'applications réparties sur une grappe de machines et gérant des données permanentes. Il offre au niveau utilisateur l'abstraction d'une MVRP dont les avantages ont été présentés précédemment. Il s'octroie pour cela une partie de l'espace d'adressage 64 bits d'AIX (adresses dans l'intervalle $[2^{63} \ldots 2^{64} - 1]$) dont il gère l'allocation de manière concurrente et répartie. De ce fait, la mémoire Arias peut être adressée de la même manière que la mémoire standard. Cette mémoire est segmentée et un segment est similaire à un segment partagé Unix.

7.2.1 Caractéristiques de la MVRP Arias

L'originalité du système Arias réside dans les caractéristiques de la mémoire qu'il offre : la permanence des données et de l'espace d'adressage, son intégration au système d'exploitation et surtout la possibilité de définir des politiques de gestions spécifiques.

7.2.1.1 Permanence

Le système met en oeuvre un espace de stockage constitué de volumes[1] répartis sur les différents noeuds de la grappe. Par défaut un segment est créé de manière *persistante*, c.-à-d. que les données qu'il contient survivent à la mort du processus créateur mais pas à l'arrêt du système. Afin de le rendre *permanent*, une image stable doit être créée dans un volume et lui être associée.

Le système offre un seul et même niveau d'adressage. En effet, une adresse virtuelle d'un segment identifie les données qu'il contient aussi bien en mémoire d'exécution que dans l'espace de stockage. Cette gestion uniforme a l'avantage de ne pas à avoir à se préoccuper des problèmes de conversion ou de correspondances d'adresses (swizzling) [Whi94, KK95]. Cette approche, souvent critiquée du fait qu'elle limite l'espace permanent à 4 giga-octets sur une architecture 32 bits, est rendue possible par les espaces d'adressage 64 bits disponibles aujourd'hui. Ceux-ci repoussent en effet cette limite au-delà de 16 milliards de téra-octets, dépassant de loin les besoins et capacités de stockage actuels.

Puisqu'une image permanente est identifiée par l'adresse virtuelle du segment qu'elle représente, il est nécessaire que l'état d'allocation de l'espace d'adressage soit conservée entre l'arrêt et le redémarrage du système. Ainsi, l'état d'allocation est

[1] Un volume regroupe un ensemble de partitions physiques d'un ou plusieurs disques connectés à une machine.

lui-même géré de manière permanente. De plus, la politique choisie par le système consiste à ne jamais réallouer une zone allouée pour un segment même si ce dernier a été détruit. Une adresse virtuelle ne peut donc identifier qu'une seule et même donnée appartenant à un seul et même segment.

7.2.1.2 Intégration

Arias se présente sous la forme d'une extension chargée, sur chaque machine, dans le noyau du système d'exploitation AIX d'IBM. Une extension est un module écrit en langage C et compilé en mode noyau qu'une commande d'administration (`kload`) permet de charger et de lier dynamiquement au coeur du système. Cette approche permet au système d'être fortement intégré au système d'exploitation et de limiter les changements de contexte comme nous l'avons présenté au chapitre 5.

AIX offre la possibilité de définir dans les extensions des procédures de traitement d'événement de pagination. L'avantage de cette approche est d'éviter des anomalies liées au phénomène de *double pagination* [GH74] qui survient lorsque le gestionnaire mémoire d'un serveur repose sur la mémoire virtuelle d'un système d'exploitation. En effet, une page virtuelle peut être représentée soit par une page en mémoire physique ou par une page disque dans un espace d'échange (swap). Le système d'exploitation implante une politique de remplacement (généralement une variante de la politique LRU) pour décider quelles pages conserver en mémoire physique. Un serveur implanté au-dessus de cette mémoire virtuelle, c'est-à-dire utilisant cette mémoire virtuelle comme représentant de son espace physique, est incapable de savoir si une page de son espace physique est représentée en fait par une page en mémoire physique ou une page disque. Cela peut être très gênant surtout si la politique de remplacement qu'il implante est contradictoire avec celle du système d'exploitation.

Dans l'exemple illustré par la figure 7.1, la mémoire allouée au serveur, représentée par les emplacements 0 à 3 contenant chacun une page de la base, est implantée à l'adresse virtuelle 0xC4000. Les emplacements étant des pages virtuelles, les données qu'ils contiennent peuvent ne pas être en mémoire primaire mais dans l'espace d'échange. C'est le cas de la page virtuelle 0xC7000 représentée physiquement par la page du swap S77. La politique de remplacement du gestionnaire mémoire, supposée être appropriée à la manière dont sont accédées les pages de la base, échoue dans sa tentative de garder la page E en mémoire primaire du fait que la politique de remplacement de la mémoire virtuelle ne le juge pas opportun. En outre, ce manque d'intégration conduit à l'encombrement inutile de l'espace d'échange puisque certaines pages peuvent avoir deux images permanentes rigoureusement exactes. C'est le cas,

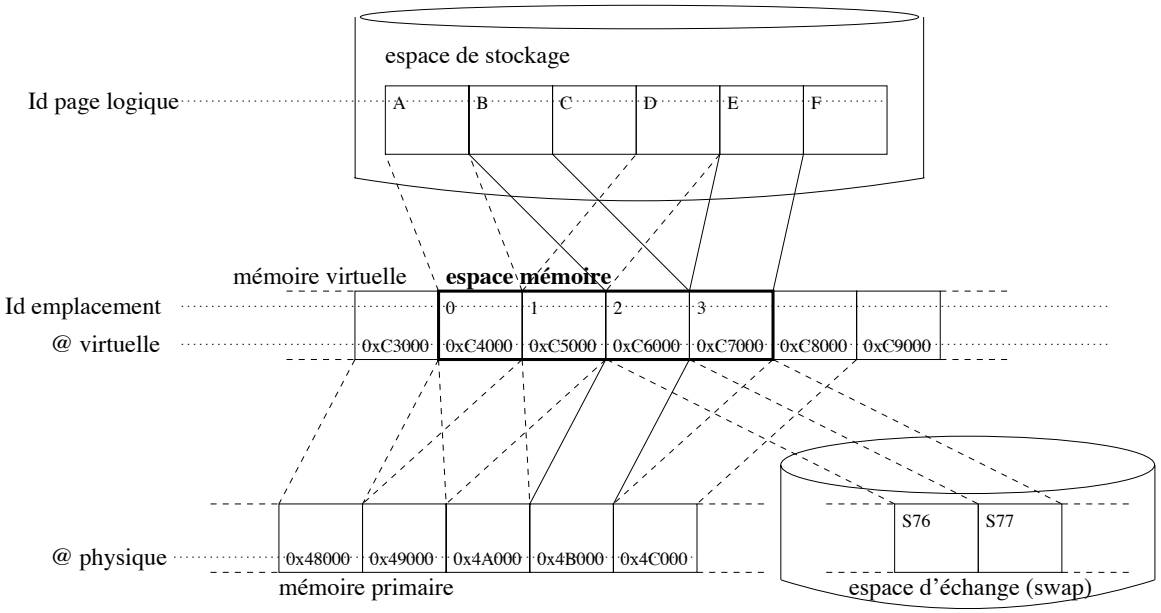


FIG. 7.1 – Anomalies liées au problème de double pagination.

dans l'exemple, de la page B qui admet une réplique (identique et donc inutile) dans l'espace d'échange (page S76). L'intégration à la mémoire virtuelle permet d'éviter ces deux problèmes à condition toutefois de se plier aux décisions de la politique de remplacement de la mémoire virtuelle.

7.2.1.3 Adaptabilité

En plus de la permanence des données et de l'espace d'adressage, la MVRP Arias possède une caractéristique qui la distingue radicalement des autres MVRP : son caractère adaptable. En effet, la plupart des MVRP offrent soit un modèle de cohérence soit une palette de modèles de cohérence parmi lesquels l'application peut choisir. L'approche choisie ici est différente puisque le système laisse à ses utilisateurs la responsabilité d'implanter leurs propres modèles de cohérence. Le système offre ainsi la possibilité de définir le modèle qui est le mieux adapté aux contraintes imposées par l'application, ni plus, ni moins.

Cette spécialisation est rendue possible par le concept de Module de Spécialisation (MS), extension noyau liée à celle d'Arias. Le MS a pour rôle de synchroniser les processus répartis s'exécutant en parallèle et de maintenir cohérentes les différentes copies d'une même page présentes dans le système. Mais ses responsabilités vont au-delà. En effet, le caractère adaptable du support ne se limite pas au modèle de cohérence mais comprend également la répartition des données sur disque, la résistance aux pannes,

la répartition de charge, la gestion de la mémoire globale, la protection, etc. Pour chacune de ces responsabilités, Arias n'impose aucun modèle, aucune politique, mais laisse cette responsabilité au MS.

Le système supporte simultanément plusieurs MS. Il est donc possible d'utiliser ce support pour gérer simultanément des espaces de données nécessitant des gestions différentes. Pour permettre cette cohabitation, les segments sont toujours créés pour le compte d'un MS et seul ce dernier peut accéder les données que contient le segment. De cette manière, le système peut servir de support à plusieurs serveurs de données s'exécutant en parallèle sur la même grappe de machine, chacun étant représenté par un MS.

Une application doit préalablement s'enregistrer comme *client* d'un MS avant d'être autorisée à accéder les données sous contrôle de ce dernier. Sans cette procédure, le système conclut à une violation de protection à la première tentative d'accès et tue le processus fautif. L'application peut néanmoins s'enregistrer auprès de différents MS et ainsi accéder à différentes bases de données en même temps.

7.2.2 Mécanismes offerts

Bien qu'il n'implante aucune politique de gestion, le système offre toutefois un certain nombre de services noyaux facilitant la manipulation des différentes ressources et directement accessibles par les modules de spécialisation.

7.2.2.1 Accès à l'espace de stockage

Le service de stockage inclut un ensemble de fonctions permettant de créer et détruire une image permanente d'un segment et d'y effectuer des entrées/sorties. Lors de la création, le MS indique le numéro de volume dans lequel doit être créée l'image, cette information étant conservée de manière permanente par le système. Pour les autres opérations, l'adresse virtuelle du segment suffit au service pour identifier et localiser de manière transparente le volume contenant son image permanente. Notons que de l'espace n'est alloué sur disque que pour les pages effectivement écrites. Cette approche permet donc de faire grandir l'image d'un segment de manière incrémentielle mais ne garantit en revanche pas que suffisamment d'espace est disponible pour créer une image complète de tous les segments. Un choix d'implantation d'Arias impose la non extensibilité des segments. Il est donc de ce fait souvent préférable de les créer avec une grande taille surtout lorsque la taille maximale des données effectivement écrites n'est pas connue à l'avance. La taille d'un segment peut donc être souvent largement

surestimée justifiant de ce fait l'approche par incréments prise pour l'allocation sur disque.

7.2.2.2 Accès à l'espace physique

Le service de gestion de la mémoire d'exécution fournit deux types de fonctions : celles permettant la gestion de la mémoire locale et celles permettant la gestion de la mémoire globale.

Du fait de sa forte intégration avec la mémoire virtuelle du système d'exploitation, Arias permet à un MS d'enregistrer des poignées afin que ce dernier puisse être notifié des exceptions de pagination concernant les pages qu'il gère. Ces événements sont :

- le défaut de page notifié lorsqu'un processus utilisateur accède à une page non résidante en mémoire d'exécution locale ;
- le remplacement d'une page notifié lorsque la mémoire virtuelle d'AIX décide de supprimer la copie locale d'une page.

Concernant la mémoire globale, le service fournit des mécanismes permettant de lire ou modifier une copie à distance. Les paramètres qui doivent être passés sont le numéro de noeud sur lequel réside la copie et l'adresse virtuelle de la page.

7.2.2.3 Résistance aux pannes

Arias offre deux services permettant de gérer la cohérence entre les données en mémoire d'exécution et celles en mémoire de stockage.

Un service de journalisation permet la création, la destruction, la modification et la validation atomique de journaux répartis. Il permet à un module de créer localement un journal et d'y inscrire des informations (valeur avant, après, information logique) concernant un ensemble de segments. Le service se charge alors de créer de manière transparente les fragments de journaux sur les différents noeuds où sont stockées les images permanentes des segments. Ce service est doublé d'un service de reprise après panne permettant de relire l'état d'un journal tel qu'il était au moment d'une panne et d'effectuer les actions nécessaires pour remettre la mémoire dans un état cohérent.

7.2.3 Architecture et gestion interne

7.2.3.1 Architecture

La figure 7.2 illustre l'architecture du système sur une machine donnée. Une instance du système Arias ainsi qu'une instance de chaque MS sont chargées dans le noyau de chaque machine.

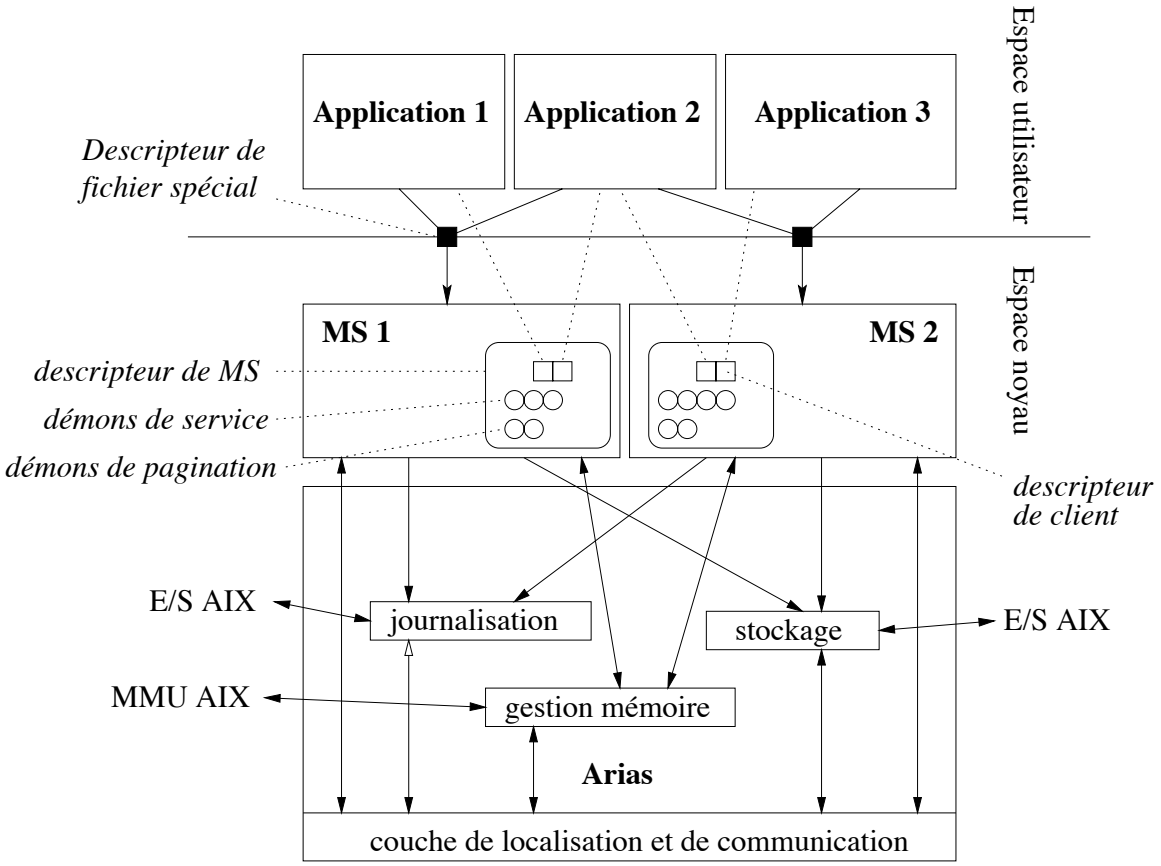


FIG. 7.2 – Architecture du système Arias.

Avant de pouvoir accéder les données gérées par MS, une application doit tout d'abord se connecter à l'instance locale de ce MS. Nous disons alors que l'application est un *client* du MS et la connexion est représentée par un descripteur de client. Une application peut se connecter simultanément à plusieurs MS mais sera représentée à chaque fois par un client différent. Inversement, un MS peut accepter plusieurs clients en même temps. De plus, une application peut se connecter plusieurs fois au même MS mais sera représentée par différents clients. De cette manière, les clients peuvent représenter différents flots d'exécution appartenant à la même application.

Les applications s'exécutent en mode utilisateur et utilisent l'appel système `ioctl` d'AIX pour communiquer avec les modules de spécialisation. Cet appel prend comme paramètres un numéro de commande (parmi un liste spécifiée par le système) et le descripteur de fichier spécial représentant le MS invoqué.

Suite à un appel système, le MS prend des décisions et invoque des services Arias, soit directement par appel de procédure synchrone, soit via le service de communication. L'appel système `ioctl` étant un appel synchrone, le MS garde un contrôle total

sur l'exécution de ses clients. Il peut en effet décider de les endormir et de les réveiller lorsqu'il le juge nécessaire.

Un ensemble de flots d'exécution, appelés *démons de services*, sont créés pour chaque MS et sont utilisés pour le traitement des appels distants aux services Arias. Deux autres démons sont également automatiquement créés par le système lors du chargement du MS afin de traiter les événements de pagination.

Sur chaque machine, le système gère des structures de données, appelées descripteurs, afin de conserver certaines informations en mémoire. On trouve essentiellement un descripteur pour chaque segment couplé localement, chaque copie résidante localement, chaque MS et chaque client. Chacun de ces structures contient un pointeur permettant d'attacher des informations supplémentaires et spécifiques au module de spécialisation.

Le support est multi-programmé puisque différents clients et démons peuvent s'exécuter simultanément dans le noyau et accéder de manière concurrente les descripteurs du système. Pour cette raison, le système met à disposition deux types de verrous : le premier permet de limiter l'accès à un descripteur par un seul client à la fois tandis que le deuxième permet de limiter l'accès à un descripteur aux clients ou démons d'un même MS.

7.2.3.2 Gestion de l'allocation et de la localisation

La mémoire virtuelle Arias est découpée en *partitions*, chaque partition étant gérée par un et un seul MS. Ainsi, tout segment créé dans une partition est sous contrôle du MS détenteur de la partition et seuls les clients de ce MS sont habilités à accéder ce segment (toute autre tentative se traduisant par un violation de segment). L'allocation de segments dans une partition est ensuite répartie sur les différents noeuds de manière à permettre le plus de parallélisme possible lors de la création et de la destruction de segments.

Le service de localisation s'appuie sur le concept de maître/esclave. Pour chaque page logique, un des noeuds parmi ceux possédant une copie de la page est appelé noeud maître de la page, les autres étant appelés noeuds esclaves. Le noeud maître conserve la liste de toutes les copies de pages et est contacté à chaque création et destruction d'une copie de la page. Les maîtrises des pages pouvant être réparties sur l'ensemble des machines, la gestion de la localisation peut également l'être, chaque noeud étant responsable de la localisation d'un ensemble de page. Cette répartition peut être de plus réorganisée à tout moment puisque le système permet la migration de la maîtrise d'une page du noeud maître vers n'importe quel noeud.

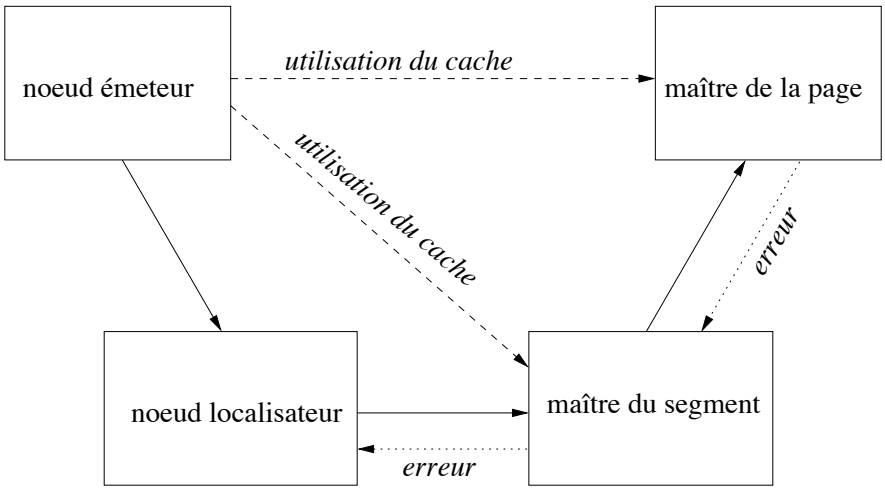


FIG. 7.3 – Localisation du noeud maître d'une page.

Lorsqu'un noeud souhaite contacter le noeud maître d'une page, le système se charge de le localiser de manière transparente. Pour cela, la notion de maître/esclave est étendue aux segments, le maître d'un segment étant chargé de maintenir la liste de tous les noeuds ayant couplés le segment (noeuds esclaves). Enfin, la localisation d'un segment s'effectue en contactant le noeud *localisateur* de la partition (c.-à-d. le noeud sur lequel a été créée la partition) contenant le segment, dont l'identité est connu de manière statique.

L'acheminement d'un message est illustré par la figure 7.3. Coûteux la première fois qu'une page d'une segment est accédée, le routage est accéléré les fois suivantes par l'utilisation d'un cache local des identifiants des noeuds maîtres des pages et des segments les plus contactés. Toutefois, les informations cachées ne sont pas mises à jour de manière synchrone lors de la migration des maîtrises et peuvent donc être temporairement inexactes. Un message destiné au maître d'une page peut donc être envoyé par erreur à l'ancien noeud de la page. Lorsque cela arrive, le message est renvoyé au maître du segment afin de retrouver la nouvelle identité du maître. De la même manière, un message envoyé par erreur à l'ancien maître d'un segment est renvoyé au site localisateur afin de retrouver le nouveau maître du segment.

7.2.3.3 Gestion des communications

Le système fournit une couche de communication afin de permettre l'appel de services sur des machines distantes ou la communication entre différentes instances d'un même MS. L'appel d'un service s'effectue par l'envoie d'un message en indiquant

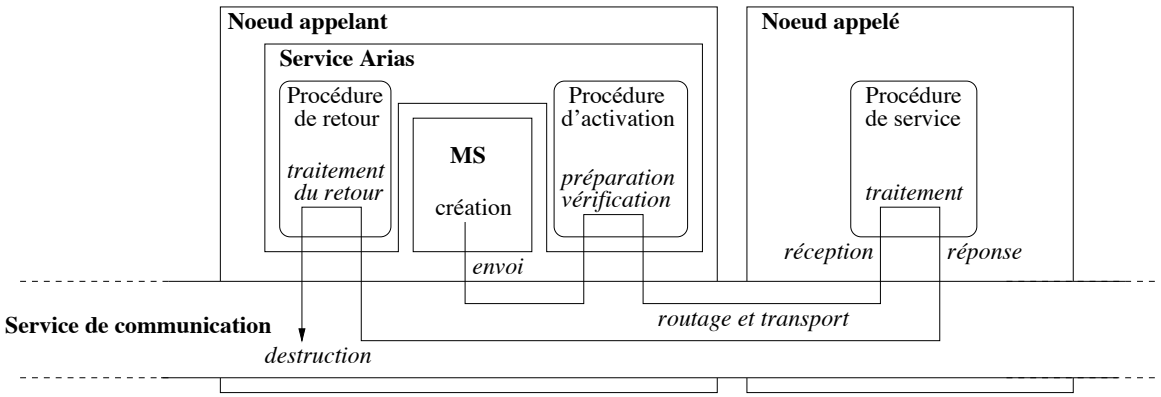


FIG. 7.4 – Différentes étapes dans le processus d'appel à un service.

l'identité du noeud destinataire et le numéro du service concerné. Le destinataire peut être explicitement désigné par son numéro de noeud ou implicitement désigné par un statut particulier tel que "maître de la page P", "maître du segment S", "noeud de stockage du segment S", etc. Dans le deuxième cas, le mécanisme de localisation du système est utilisé afin d'acheminer le message à bon port.

L'appel à un service s'effectue en trois étapes tel que représenté par la figure 7.4. Lors de son initialisation, un service Arias enregistre 3 procédures : une procédure d'activation, une procédure de traitement, et une procédure de retour. Ces procédures sont invoquées successivement par le service de communication lors de l'appel d'une de ses fonctions. Suite à l'invocation de la procédure `ReqActivate`, le message est passé à la procédure d'activation correspondant au service appelé. Celle-ci se charge de vérifier la validité des paramètres passés et de compléter le message si nécessaire. Le message est ensuite acheminé vers le bon noeud où la procédure de traitement du service est appelée. Les actions adéquates (E/S, copie mémoire, validation d'un journal, etc.) sont alors effectuées et un message contenant le résultat est retourné au noeud appelant. Le résultat est ensuite passé à la procédure de retour qui effectue les modifications nécessaires (création d'une copie locale, modification du contenu, etc.).

Afin de permettre le contrôle total de l'invocation d'un service, la couche de communication permet la surcharge de n'importe quel service. Comme le montre la figure 7.5, le module de spécialisation peut enregistrer pour chaque service des procédures de contrôle qui seront invoquées avant et après les procédures de traitement et de retour du service spécifié. Celles-ci peuvent alors effectuer des actions complémentaires voire, pour la procédure *pré-service*, décider d'annuler l'appel.

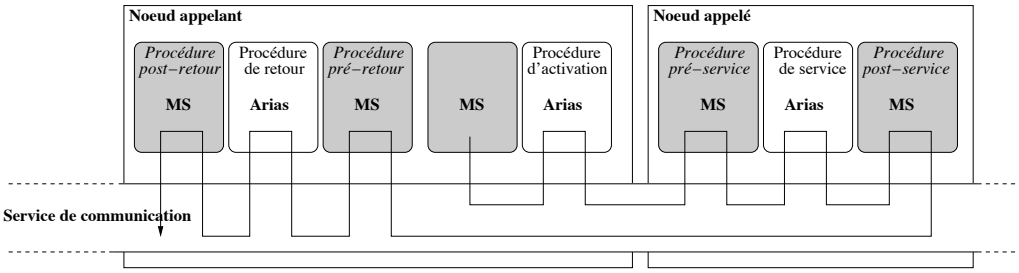


FIG. 7.5 – Surcharge d'un appel à un service par le module de spécialisation.

7.2.3.4 Détection des accès

L'architecture du système permet deux modes de détection d'accès à la mémoire par les processus utilisateurs. Le premier mode consiste à informer explicitement le système via un appel système que l'application désire accéder une page ou partie d'une page. Le MS intercepte cet appel pour effectuer le contrôle d'accès et mettre en cohérence la copie locale et éventuellement la charger du disque ou d'une mémoire distante. Le deuxième mode consiste à détecter les accès en récupérant et traitant de manière implicite les événements de défaut de page. Cette approche simplifie le code de l'application qui n'a alors plus aucune distinction entre la mémoire Unix et la mémoire Arias. Elle est cependant insuffisante lorsque par exemple un partage à grain inférieur à celui de la page est nécessaire.

7.3 Implantation d'ADAMS

L'implantation d'un serveur de données réparti avec Arias nécessite la programmation d'un module de spécialisation puis son chargement dans le noyau. En supportant le chargement de plusieurs modules de spécialisation, le support peut être utilisé simultanément pour différents serveurs offrant l'accès vers différentes bases de données. Son apport pour les serveurs de données est cependant limité pour les raisons suivantes :

- Bien qu'il permette d'implanter des politiques de cohérence et de synchronisation spécifiques, il n'en est pas de même pour les politiques de remplacement et d'allocation.
- Changer le comportement d'un serveur nécessite de remplacer son module de spécialisation rendant totalement indisponible le serveur pendant cette période.
- Aucun support n'est fourni pour la hiérarchisation, l'intégration et l'interaction de différentes politiques de gestion et donc pour minimiser les dépendances entre

les politiques.

Afin de combler ces lacunes nous avons étendu et modifié le système afin de mettre en oeuvre le support mémoire ADAMS introduit au chapitre précédent. Après avoir présenté les contraintes imposées par le contexte d'implantation, nous montrons dans cette section comment nous avons implanté ses différents services ainsi que les structures de données qu'ils utilisent.

7.3.1 Contraintes et choix d'implantation

7.3.1.1 Gestion des contextes

Nous utilisons la notion de client comme représentation des traitements. L'identifiant d'un client est constitué par l'identifiant du MS pour lequel il a été créé, l'identifiant du noeud sur lequel il s'exécute et un numéro le différenciant des autres clients du MS sur le même noeud. Typiquement, un MS représentera un serveur de données et un client représentera une requête du serveur. Chaque partition ne pouvant être accédée que par les clients du MS auquel elle est rattachée, chaque serveur gère de manière indépendante sa propre base de données. Notons que cette contrainte propre ne permet pas à deux serveurs de partager des données. Toutefois, la notion de serveur dont nous parlons ici correspond à une entité de gestion des ressources. Rien n'empêche de bâtir deux services partageant des données gérées par deux serveurs différents.

Nous représentons chaque MS par un contexte responsable de la gestion des accès effectués par ses clients sur des segments appartenant aux partitions rattachées au MS. Lorsque l'accès à un segment est signalé ou détecté, le support est capable de retrouver directement le contexte du MS à partir soit de l'identifiant du client (qui contient l'identifiant du MS) soit à partir de l'identifiant du segment (qui contient l'identifiant de la partition) et de la table de partition (contenant pour chaque partition l'identifiant du MS auquel elle est rattachée).

Un contexte global est automatiquement créé à l'initialisation du support et sert essentiellement à gérer l'allocation entre les différents serveurs. Au moment de son chargement, un MS doit indiquer les valeurs d'allocation minimale et maximale qu'il désire qui sont accordées selon les conditions imposées par le modèle. Ces valeurs sont ensuite utilisées pour initialiser le contexte correspondant au MS.

Dans le modèle, l'espace de gestion d'un contexte est représenté par un ensemble de couples $\{traitement, page\}$. Cependant, ce grain de définition est bien trop fin dans la plupart des cas et nécessiterait des structures de données bien trop larges et des fonctions de recherche bien trop coûteuses. Pour réduire ce coût nous construisons

les espaces de gestion en nous appuyant sur les notions de segment, de client et de partition. Chaque espace de contrôle est défini par restriction de l'espace de contrôle de son père par rapport à un ensemble de segment, un ensemble de partitions ou un ensemble de clients. Cette manière de définir les espaces de contrôle est justifiée par les raisons suivantes :

- subdiviser les contextes selon les partitions, les segments ou les clients est justifié du fait que les contextes sont généralement subdivisés selon les données ou selon les traitements mais rarement selon les deux,
- choisir le segment comme unité de différenciation des données se justifie par le fait qu'il permet de représenter une donnée constituée d'un ensemble de pages,
- enfin, permettre de subdiviser les contextes selon les partitions se justifie par le fait que ces dernières peuvent être utilisées pour représenter des domaines de données (index, relations, etc.).

Comme nous le verrons plus loin, définir les espaces de gestion de la sorte permet d'implanter des méthodes de recherche de contexte efficaces sans limitation des possibilités de hiérarchisation.

7.3.1.2 Gestion des copies et emplacements

Le fait de s'appuyer sur la mémoire Arias permet de simplifier considérablement les problèmes d'adressage entre les différentes machines et entre la mémoire d'exécution et l'espace de stockage. Cependant, cette mémoire impose la contrainte qu'une page ne peut être répliquée sur un même noeud puisque le mécanisme de correspondance entre l'espace physique et l'espace logique est réalisé entièrement par le système d'exploitation. Ce dernier reste en effet maître de la correspondance entre les emplacements et les pages virtuelles. Lorsqu'une page est chargée en mémoire, il décide quel emplacement doit être utilisé. Les mécanismes d'adressage matériels et logiciels assurent ensuite de manière transparente la correspondance entre les adresses virtuelles manipulées par le processeur et les pages physiques contenant les données. L'impossibilité de manipuler directement les identifiants des emplacements n'est cependant pas gênante du fait que la répartition des emplacements entre les différents contextes est effectuée en terme de nombre d'emplacements.

Le système d'exploitation possède sa propre politique de remplacement et ne permet pas son adaptation. Pour cette raison, les copies pour lesquelles un lien primaire existe sont fixées en mémoire par le support à l'aide de la fonction `pin` mise à disposition par le système d'exploitation (ce dernier ne peut donc plus les remplacer). Lorsqu'un lien primaire est changé en lien secondaire, la copie correspondante est

dé-fixée à l'aide de la fonction `unpin` également disponible. Celle-ci peut alors être remplacée à tout moment par la politique de remplacement du système d'exploitation. L'intérêt, cependant, est que les copies remplacées sont conservées dans l'espace d'échange (swap) du système d'exploitation et que ce dernier gère de manière complètement transparente la correspondance entre la page virtuelle et l'emplacement sur disque. Si aucune des politiques de remplacement ne décide de conserver la copie mais que les données doivent cependant être conservées, il convient simplement de ne pas détruire la copie. Si cette dernière a été remplacée par le système d'exploitation, elle est automatiquement et de manière transparente chargée à nouveau en mémoire à son prochain accès.

Le support doit pouvoir garantir de toujours pouvoir fixer les pages en mémoire dans la limite de la valeur d'allocation du contexte général. Afin de s'assurer que le système d'exploitation ne lui refusera une telle opération, il fixe dès le départ un ensemble de pages dans un segment spécial, appelé *segment d'allocation*. La création d'un lien primaire consiste alors à fixer la copie en mémoire et dé-fixer une des pages du segment d'allocation. Inversement, la destruction d'un lien primaire revient simplement à dé-fixer la copie correspondante et fixer une page du segment d'allocation.

7.3.2 Priorité d'optimisation des fonctions du support

Le support doit être implanté de telle manière que son coût d'utilisation soit le plus faible possible. Le sur-coût induit par l'utilisation d'un fonction du support est d'autant plus acceptable si l'utilisation de la fonction est peu fréquente. D'une manière générale, il convient donc d'optimiser en priorité les fonctions qui sont le plus utilisées. Ensuite, le coût d'utilisation d'une fonction est acceptable s'il est faible par rapport aux coût annexes liés à l'utilisation de la fonction. Les fonctions spécifiques de notre support sont celles permettant l'ajout de nouvelles politiques dans le système, la création de contexte et le rattachement de politiques puis la notification d'événements.

Ajout de politiques La nécessité d'introduire de nouvelles politiques dans le support est nécessaire lorsque les politiques déjà chargées ne permettent pas de prendre en compte de nouveaux besoins. De même, la création de nouveaux types d'événements n'est nécessaire que lorsque de nouveaux changements d'états du système doivent pouvoir être pris en compte. Ces situations sont a priori rares et font suite à une étude préalable et à l'implantation et la compilation dont le coût en temps est de toute façon élevé. En conséquence, la minimisation du coût des fonctions d'enregistrement de politique, de création d'événements et d'abonnement des politiques aux événements n'est pas une priorité. L'essentiel

est plutôt de permettre l'intégration de nouvelles politiques en limitant l'impact sur la disponibilité du serveur.

Rattachement de politiques Le rattachement d'une politique à un contexte doit être effectué à chaque fois qu'un nouveau contexte est créé. Il s'agit donc d'une opération bien plus fréquente puisque de nouveaux contextes peuvent par exemple être créés pour chaque nouveau traitement voire à chaque fois qu'un parcours d'un nouvel ensemble de données est détecté. Il est par conséquent nécessaire que les fonctions de création de contexte et de rattachement de politique aient un coût faible par rapport au temps d'exécution des traitements.

Notification d'événements La notification d'événements est de loin ce qu'il y a de plus fréquent. En effet, des événements sont signalés à chaque accès, chaque défaut de page, chaque fois qu'un emplacement doit être libéré, etc. De plus, chacun de ces événements peut toucher plusieurs politiques. Le coût de la fonction de notification d'événements doit impérativement être réduit au minimum et doit en particulier être faible par rapport au temps de réaction aux événements.

7.3.3 Gestion des politiques et des événements

La gestion de politiques et d'événements constitue le coeur du support. Cette section décrit les principales fonctions correspondant à cette gestion et notamment comment les structures de données représentées dans la figure 7.6 permettent de minimiser le coût d'utilisation de ces fonctions et de répondre aux exigences présentées dans la section précédente.

7.3.3.1 Ajout de politiques

Tout comme un MS, une politique se présente sous la forme d'une extension noyau que la commande d'administration `poladm` permet de charger et décharger dans le noyau. Afin d'être utilisable par les différents modules de spécialisation, la politique doit s'enregistrer auprès d'ADAMS à l'aide de la fonction `enregistrer_politique` prenant comme argument une chaîne de caractère représentant le nom de la politique et retourne l'identifiant de la politique. Comme nous l'avons expliqué au chapitre précédent, une politique doit être chargée et enregistrée sur chaque noeud où elle est susceptible d'être utilisée. Cependant, ADAMS assure une correspondance unique entre le nom d'une politique et son identifiant en répliquant la table de correspondance `TPol` sur tous les noeuds de la grappe. Étant donné la fréquence d'enregistrement des politiques, la non scalabilité de cette approche n'est heureusement pas un problème

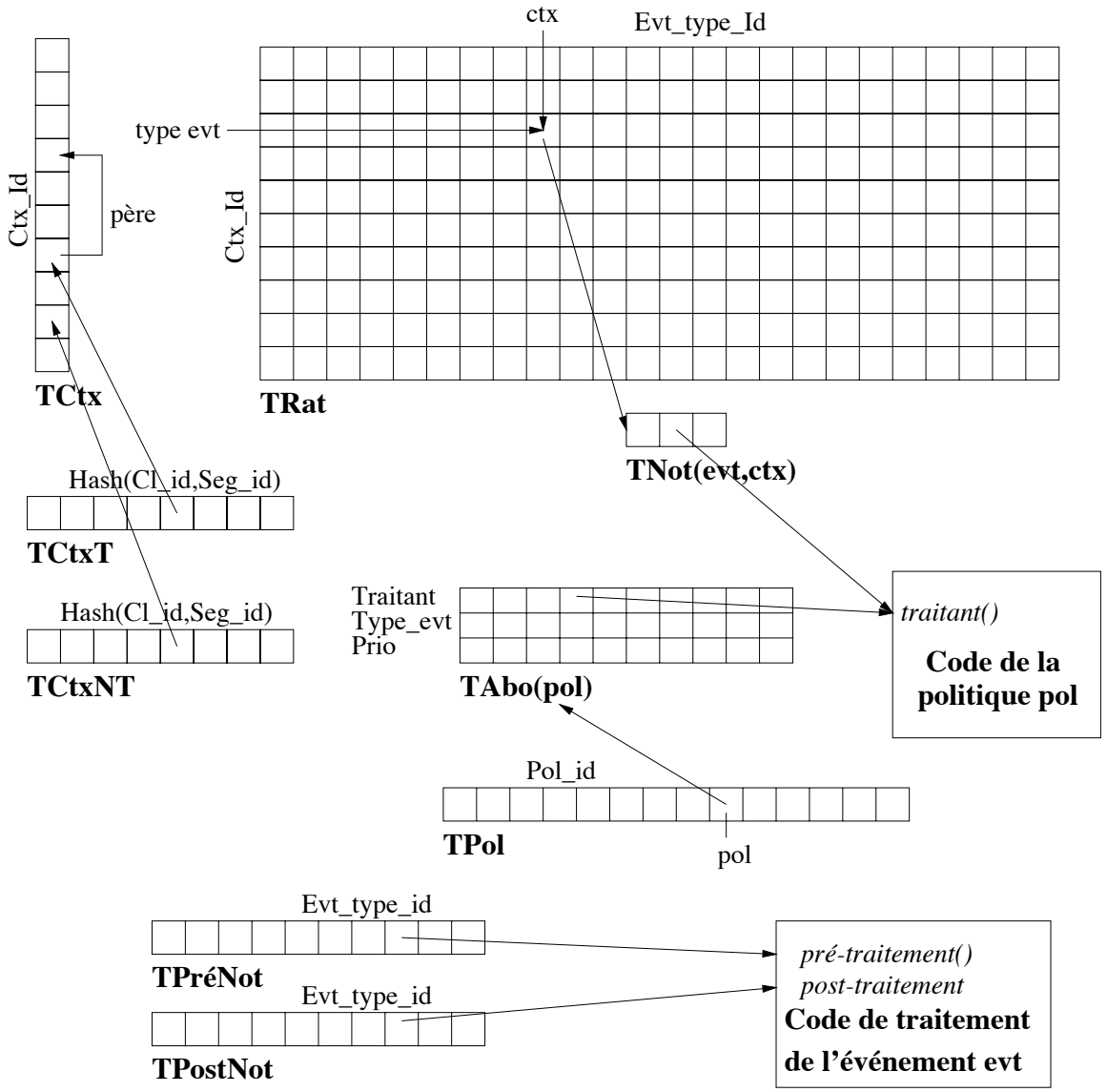


FIG. 7.6 – Structures de données utilisées pour la gestion des politiques et des événements.

significatif.

Une fois chargée, la politique est sensée s'abonner aux types d'événements qui la concerne grâce à la fonction `abonner_politique` qui prend comme argument l'identifiant de la politique et le type d'événement, la priorité d'appel (`en_premier`, `en_dernier`, etc.) et un pointeur vers la fonction de traitement de l'événement. ADAMS offre une fonction permettant d'établir la correspondance entre l'identifiant d'un type d'événement et le nom qui lui a été attribué lors de sa création. Toutefois, les identifiants des types d'événements pré-définis par le support sont des constantes qui peuvent être directement utilisées lors de l'implantation de la politique. Le support maintient pour chaque politique, une table `TAbo` dont chaque entrée contient l'identifiant d'un type d'événement, la priorité d'appel et l'adresse de la fonction de traitement.

7.3.3.2 Rattachement de politiques

Une fois qu'une politique a été chargée, s'est déclarée auprès du support et s'est abonnée elle peut être rattachée à un contexte à l'aide de la fonction `rattacher_politique` prenant comme argument l'identifiant de la politique et celui du contexte concerné. Cette fonction a notamment pour tâche de vérifier que pour chaque type d'événement auquel la politique est rattachée, la priorité d'appel est compatible avec celle des autres politiques déjà enregistrées. Pour cela, elle s'appuie et met à jour la table `TRat` qui contient une entrée pour chaque type d'événement et pour chaque contexte. Chaque entrée contient un pointeur vers une table `TNot` contenant les pointeurs vers les fonctions de traitement concernées, un pointeur nul indiquant qu'aucune des politiques rattachées au contexte n'est concernée par l'événement. Les pointeurs de fonction des tables `TNot` sont ordonnées en tenant compte de l'ordre de priorité d'appel.

Étant donné la fréquence de création de contexte (et donc de rattachement de politique), le support offre la possibilité de définir des *groupes de politiques*. La fonction `créer_groupe` renvoie un identifiant correspondant à un contexte fictif qui peut ensuite être utilisé par la fonction `rattacher_politique` afin d'ajouter des politiques à ce groupe et de vérifier la compatibilité des politiques. Une fois que le groupe est complet, la fonction `rattacher_groupe` permet de rattacher d'un seul coup et autant de fois que nécessaire un ensemble des politiques à un contexte sans avoir à vérifier leur compatibilité. Le coût de cette fonction est uniquement la copie de la ligne d'entrée dans la table `TRat` correspondant au contexte fictif vers la ligne correspondant au contexte auquel est rattaché le groupe de politiques.

7.3.3.3 Notification d'événements

La notification d'un événement nécessite de retrouver les contextes concernés avant de pouvoir effectivement notifier l'événement aux politiques abonnées. Pour certains types d'événements comme `evt_contexte_activé`, `evt_contexte_désactivé`, `evt_emplacement_alloué` ou `evt_emplacement_libéré`, un seul contexte est concerné qui peut être retrouvé facilement et rapidement. Le problème se pose plutôt pour les événements `evt_début_accès` et `evt_fin_accès` qui doivent être notifiés à tous les contextes concernés par l'accès.

Une première solution est de partir du contexte global et de rechercher de manière récursive le contexte fils qui est concerné par l'appel. Cette méthode nous semble cependant trop coûteuse étant donné la fréquence de notification de tels événements. L'approche que nous avons choisie est une approche ascendante. Le support maintient en effet sur chaque machine une table de hachage `TCtxT` dont la fonction d'indexation prend comme paramètres l'identifiant du client effectuant l'accès et l'adresse du segment accédé. Cette table donne en sortie l'identifiant du contexte terminal concerné par l'appel. Comme le support maintient dans le descripteur de chaque contexte l'identifiant de son père, il est capable de retrouver l'ensemble des contextes concernés par l'appel. Une autre table de hachage, appelée `TCtxNT`, permet de retrouver l'ensemble des contextes non terminaux à partir de l'identifiant d'un client et l'adresse d'un segment. Cette table est notamment utilisée lorsqu'il n'existe pas encore de contexte terminal capable de gérer l'accès et permet de notifier l'événement `evt_pas_de_contexte_terminal`.

En plus de retrouver les contextes auxquels doit être notifié un événement, le support doit retrouver pour chaque contexte l'adresse des fonctions de traitement des politiques abonnées à cet événement. Fort heureusement, les tables `TRat` et `TNot` permettent de retrouver très rapidement ces fonctions. La fonction `notifie_événement` réalise cette tâche. Comme indiqué dans l'algorithme 3, elle appelle tout d'abord la procédure de pré-traitement puis récupère à partir de la table `TRat` l'adresse de la table `TNot` contenant l'ensemble de fonctions à appeler. Elle parcours ensuite cette table afin d'appeler chacune des fonctions avant d'invoquer la fonction de post-traitement.

7.3.4 Gestion de la localisation

Nous avons modifié le gestionnaire de localisation des copies réparties offert par le système Arias et décrit dans la section 7.2.3.2 afin de prendre en compte la notion

Algorithm 3 Fonction `notifie_événement`.

```
notifie_evt(evt_type, ctx, * args) {
  TPréNot[evt](ctx, args, * resultat)
  si résultat->action = evt_routé
    signale_evt(evt_type, résultat->ctx_dest, args)
  sinon-si résultat->action = evt_terminé
    retourner

 TNot = TRat[evt_type, ctx]
  pour chaque f dans TSig {
    f(evt_type, ctx, args, * resultat)
    si resultat = evt_terminé sort du pour
}
  TPostNot[evt](contexte, args, * resultat)
  si résultat->action = evt_routé
     notifie_evt(evt_type, résultat->ctx_dest, args)
  sinon-si résultat->action = evt_terminé
     retourner
}
```

de version. Le maître d'une page stocke l'identité des noeuds esclaves ainsi que le numéro de version de la page qu'ils possèdent. Les numéros de version attribués aux différentes copies sont choisis sur le noeud maître par la politique de C&S. Seul le noeud maître est capable d'identifier exactement l'ensemble des copies présentes dans le système. Toutefois, déduire si une page est répliquée ou non peut toujours être effectué localement à un noeud. En effet, soit le noeud est maître de la page auquel cas il connaît le nombre exact de copies de la page dans le système, soit il est esclave et il peut être certain qu'il existe une autre copie sur le noeud maître. Ceci est particulièrement important pour les politiques de remplacement réparties pour lesquelles la décision de supprimer localement une copie dépend uniquement de sa réplication et non de l'identité des noeuds qui possède une copie. Le système génère l'événement `evt_change_réplication` dans les cas suivants :

- le noeud est maître, une copie esclave est créée alors qu'il n'en existait pas,
- le noeud est maître et la dernière copie esclave vient d'être détruite,
- le noeud obtient la maîtrise alors qu'il existe encore d'autres copies esclaves.

De cette manière, les politiques (essentiellement de remplacement) peuvent être averties lorsqu'une copie passe de l'état répliquée à l'état non-répliquée et inversement.

Au besoin, le gestionnaire de localisation peut retourner la liste des noeuds possédant une copie de la page en contactant le noeud maître.

7.4 Exploitation du support

Afin de valider notre approche, nous montrons dans cette section comment nous avons exploité le support ADAMS pour implanter un serveur transactionnel. Dans un premier temps, nous présentons une implantation des différentes politiques de gestion propres au serveur et nous décrivons les interactions entre le serveur et le support. Dans un deuxième temps, nous montrons comment le caractère adaptatif du support nous permet de faire évoluer le serveur vers une deuxième version intégrant de nouvelles qualités de service tout en minimisant l'impact sur la disponibilité du serveur. Le serveur est implanté sur une grappe de machines dans le contexte du système Arias. L'implantation du support ADAMS qu'il utilise est par conséquent celle que nous avons présenté précédemment.

7.4.1 Implantation d'un serveur transactionnel

Le serveur que nous présentons ici permet d'exécuter des transactions sur une base de données relationnelle. Les relations de la base ainsi que les fichiers d'index sont stockées dans des segments Arias. Grâce aux caractéristiques de ce dernier, la gestion des références nécessaires à la mise en oeuvre des index est simplifiée au maximum puisqu'elle peuvent être représentées par des adresses virtuelles permanentes. Chaque transaction est représentée par un client Arias, représenté lui-même par un descripteur auquel le serveur rattache des informations relatives à l'exécution de la transaction (par exemple la liste des verrous qu'elle possède).

La gestion de la mémoire du serveur est une combinaison de différentes politiques qui ont été présentées dans les chapitres précédents : la politique de cohérence et synchronisation "Adaptive Call-Back Locking", la politique d'allocation "Marginal Gains", et la politique de remplacement répartie GMS. L'organisation des contextes du serveur sur une machine donnée est illustrée par la figure 7.7. Un module de spécialisation est initialement chargé sur chaque machine. L'ensemble des modules de spécialisation coopèrent de manière à former le gestionnaire mémoire réparti du serveur. Nous supposons que le serveur que nous décrivons est le seul à s'exécuter sur la grappe et récupère par conséquent la totalité des ressources mémoires disponibles. Un fois chargé, le module crée le contexte M dont l'espace de gestion regroupe l'ensemble des accès effectués sur sa machine (tous les clients x tous les segments). L'intérêt

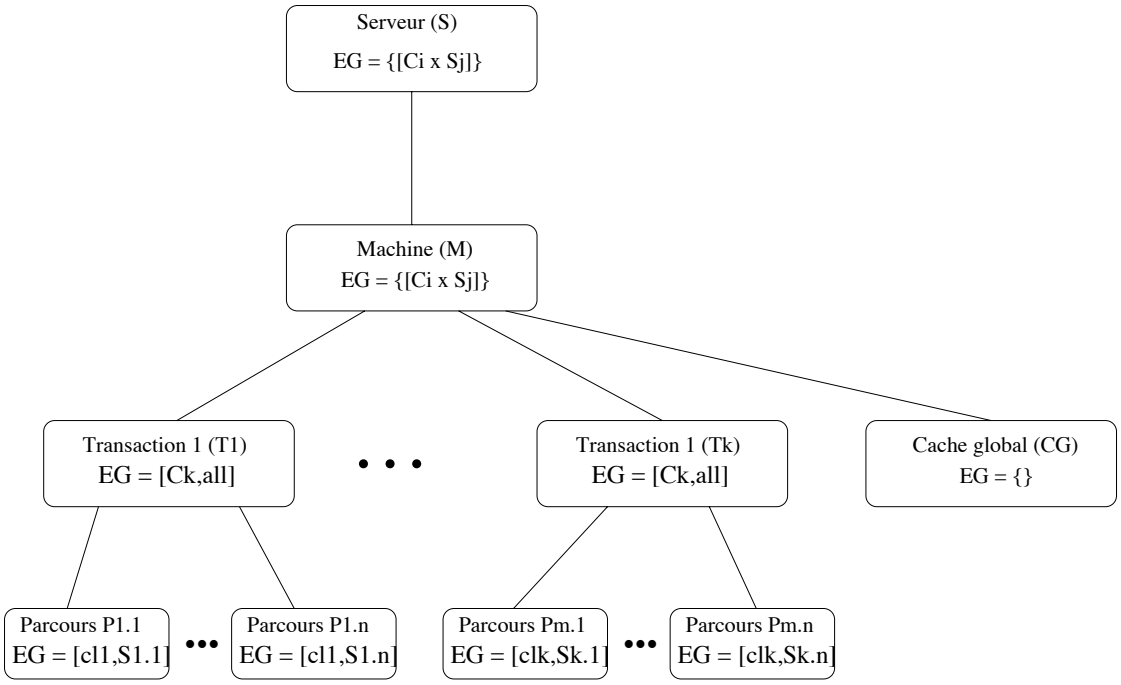


FIG. 7.7 – Organisation du serveur implanté.

de créer ce contexte plutôt que de le confondre avec celui représentant le module de spécialisation (S) sera justifié dans la section suivante. Un contexte CG est créé avec un espace de gestion vide et sert à préserver des pages utiles sur d'autres machines mais non accédées localement. Enfin, pour chaque transaction T_i (représentée par un client C_i) un contexte est créé et des sous-contextes sont créés pour chacune des relations qu'elle parcours. Contrairement au contextes CG, M et S qui sont créés une fois pour toutes, ces contextes sont créés et détruits dynamiquement à mesure que les transactions entrent et quittent le système.

Notons que si le serveur permet l'exécution en parallèle de transactions sur différentes machines il ne permet cependant pas l'exécution de transactions réparties. En effet, lors de sa création une transaction est affectée une fois pour toute à une machine par un service externe au serveur.

7.4.1.1 Politique de cohérence et synchronisation

Une transaction est représentée par un client Arias et un contexte lui est associé par le gestionnaire lors de la notification de l'événement `evt_début_traitement`. Le serveur assure l'isolation des transactions par l'intermédiaire de la politique de cohérence et synchronisation Adaptive-Call Back Locking (ACBL) présentée dans

[Fra96]. La politique que nous avons choisit d'implanter en diffère cependant dans la mesure où elle permet de capitaliser l'obtention d'un verrou par une transaction pour l'ensemble des transactions qui s'exécutent sur la même machine. De plus, le service de gestion de maîtrise nous permet de répartir dynamiquement la gestion des verrous sur l'ensemble de la grappe et de réduire l'occupation mémoire puisque la notion de "cache serveur" n'existe plus.

La politique se découpe en deux niveaux dans lesquels sont gérés deux types de verrous différents. Comme le montre la figure 7.8, les verrous globaux sont gérés par la politique de C&S globale (CSG) et permettent le partage d'une donnée avec les autres machines de la grappe. Les verrous locaux permettent quant à eux la gestion du partage entre deux transactions s'exécutant sur une même machine et sont gérés par la politique de C&S locale (CSL). La politique CSG est rattachée au contexte M alors que la politique CSL est rattachée aux contextes T_i représentant les transactions. Elles sont toutes deux abonnées au type d'événement `evt_début_accès` afin d'être notifiées des accès effectués par les transactions. Ce type d'événement étant de nature descendante, il est d'abord notifié à la politique CSG avant d'être notifié à la politique CSL.

Avant d'accorder à une transaction T s'exécutant sur une machine M l'autorisation d'accéder une donnée, la politique CSG obtient dans un premier temps un verrou global suffisant en déroulant l'algorithme ACBL. Si le verrou global possédé par la machine M est insuffisant, une demande est envoyée au maître de la page concernée qui transmet à son tour une demande de rapatriement de verrous aux machines qui possèdent un verrou global conflictuel. Une fois les verrous rapatriés, le maître envoie le verrou demandé vers la machine M. Éventuellement, la maîtrise de la page est jointe si le gestionnaire mémoire constate que seule la machine M possède des verrous globaux sur la page.

Pendant tout le processus d'obtention d'un verrou global, l'événement `evt_début_accès` est bloqué par la politique CSG. La politique CSL n'est avertie de l'occurence de cet événement que lorsqu'un verrou global suffisant a été obtenu. En fait, la gestion des verrous globaux lui est complètement inconnue ce qui facilite l'indépendance de ces deux politiques. Un verrou local est alors obtenu pour le compte de la transaction. Si le type d'accès effectué est compatible avec les autres transactions locales le verrou lui est accordé immédiatement sinon l'événement est bloqué jusqu'à terminaison des transactions conflictuelles.

Les CSG et CSL réagissent également aux événements de type `evt_fin_traitement` notifié lors de la terminaison des transactions. Dans le cas des politique de C&S basées sur des techniques de verrouillage, une transaction demandant sa validation est auto-

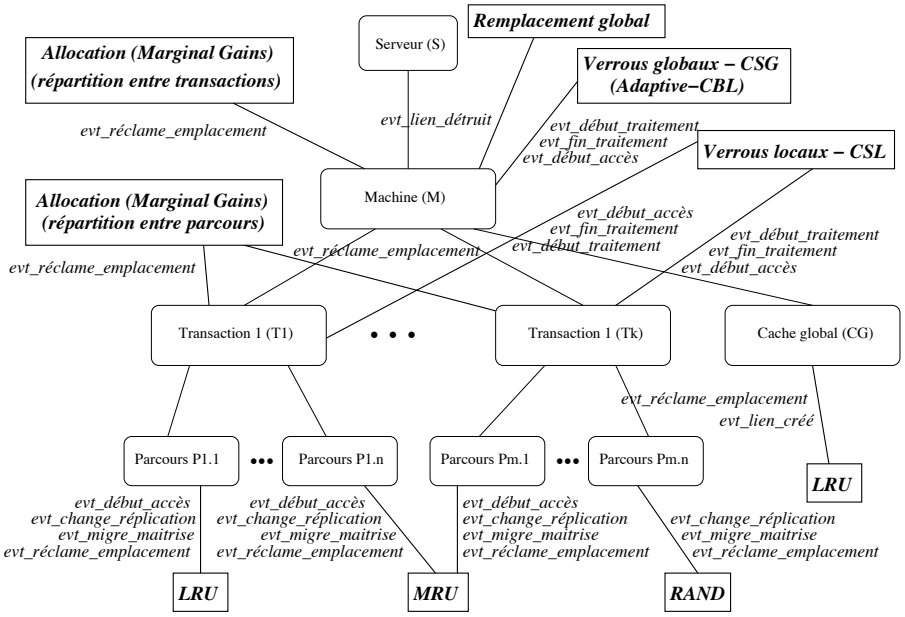


FIG. 7.8 – Rattachement des politiques aux contextes du serveur.

matiquement validée puisqu'elle est sérialisable de par la nature de la politique. Les verrous locaux sont alors relâchés par la politique CSL tandis que la politique CSG ne libère les verrous globaux que lorsqu'ils ne sont plus utilisés localement et qu'une demande de rapatriement a été enregistrée. Notons que contrairement à une politique de C&S non transactionnelle qui se doit de réagir à l'événement `evt_fin_accès`, les politiques de notre serveur l'ignorent toutes deux. Cet événement leur est en effet indifférent du fait qu'elles respectent le protocole de verrouillage à deux phases.

7.4.1.2 Politique d'allocation

La politique d'allocation choisie afin de répartir les emplacements entre les différentes transactions est la politique Marginal-Gains présentée dans le section 4.2.4. Un contexte est créé pour chaque transaction puis des sous-contextes terminaux sont créés pour chaque parcours de relation par une transaction donnée. La politique de remplacement est associée à un contexte terminal et le nombre minimal et maximal d'emplacements est choisi en fonction du parcours qui va être effectué. L'information sur le type de parcours est indiquée via une interface propre au module de spécialisation[2] par le gestionnaire d'exécution des requêtes s'exécutant en mode utilisateur.

La politique d'allocation est divisée en deux politiques séparées. L'une, respon-

[2]Le système Arias permet en effet à tout module de spécialisation de spécialiser son interface afin de permettre des échanges spécifiques avec le monde extérieur.

sable de la répartition des emplacements entre les différentes transactions, est rattachée au contexte M. L'autre, responsable de la répartition entre différents parcours des emplacements allouée à une transaction est rattachée aux contextes T_i. Pendant l'exécution, les deux politiques réagissent à l'événement `evt_réclame_emplacement` afin de répondre aux demandes d'extension des espaces d'allocation des contextes associés respectivement aux différentes relations ou aux différents parcours.

La répartition des emplacements entre les transactions repose sur l'équité : le nombre d'emplacements alloués à une requête est égal au nombre d'emplacements total divisé par le nombre de transactions. Au sein d'une transaction les valeurs d'allocation sont identiques pour chaque parcours : la valeur minimale est égale à 1 tandis que la valeur maximale est le nombre de pages de la relation. Le nombre effectif d'emplacements alloués n'est cependant pas équitable. En effet, compte tenu de la différence des gains occasionnés par l'adjonction d'un emplacement supplémentaire selon le type de parcours, la politique choisit de privilégier dans l'ordre les parcours MRU, LRU puis RAND.

Notons qu'à deux parcours effectués par deux transactions différentes sont associés deux contextes différents même si ces parcours portent sur la même relation. Cela n'induit pas de duplication de données (puisque notre modèle permet le partage d'une page par deux contextes différents) et permet d'appliquer deux politiques de remplacement différentes aux pages d'une même relation.

7.4.1.3 Politique de remplacement

Une politique de remplacement (LRU, MRU ou RAND) est donc associée à chaque parcours de relation par une transaction donnée. Afin de bénéficier des avantages offerts par la plate-forme, un contexte est créé sur chaque machine dans le but de préserver, lorsque cela est possible, les pages rejetées par des machines distantes. Les valeurs d'allocation attribuées à ce contexte sont min = 0 et max = la taille de la mémoire locale. Les emplacements sont attribués en priorité aux contextes des transactions locales de sorte que le contexte CG ne récupère que les emplacements non utilisés localement. Comme le montre la figure 7.9, la politique de remplacement globale correspond ainsi à la politique présentée dans [Fra96] où la mémoire d'une machine est divisée en un cache local des pages accédées localement et un cache global des pages utiles sur les autres machines et dont la frontière varie dynamiquement. Dans notre cas, le cache local est lui-même subdivisé en autant de sous-caches que transactions s'exécutant localement, chacun de ces sous-caches étant à son tour subdivisé en autant de sous-caches que de relations parcourues par la transaction

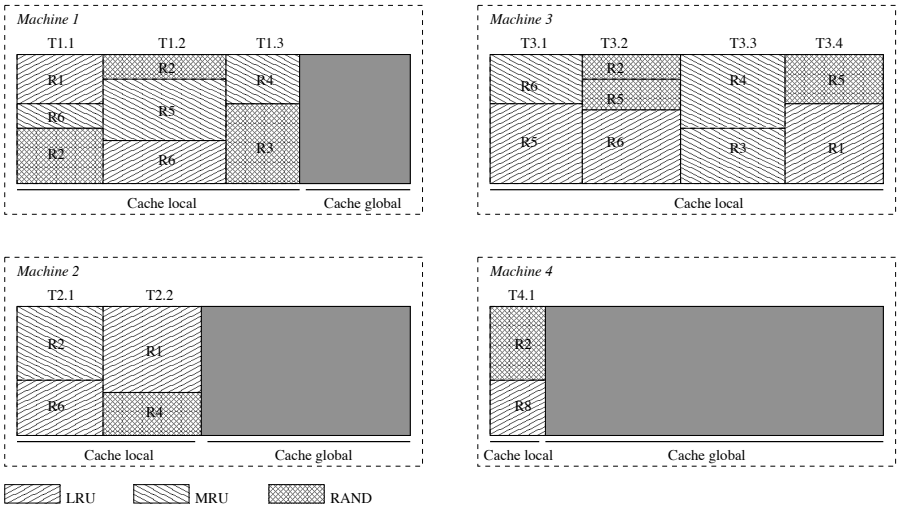


FIG. 7.9 – Subdivision de la mémoire dans le serveur implanté.

correspondante.

La politique LRU maintient, pour chaque contexte auquel elle est associée, trois listes lui permettant d'estimer l'utilité des pages accédées dans ce contexte. La première est la liste des pages répliquées dont la maîtrise est distante, la deuxième est la liste des pages répliquées dont la maîtrise est locale et la troisième est la liste des pages non répliquées (et dont la maîtrise est par conséquent locale). Ces trois listes sont gérées de manière LRU : lorsque l'événement `evt_début_accès` lui est notifié, la page correspondante est positionnée à la fin de la liste à laquelle elle appartient. La politique réagit également aux événements du type `evt_change_réplication` afin de mettre la page concernée dans la liste qui lui convient. Afin d'être informé des mouvements de maîtrise, la politique de C&S définit le nouveau type d'événement `evt_migre_maîtrise` qui est notifié à chaque migration de maîtrise sur les noeuds de départ et d'arrivée. La politique LRU réagit à cet événement afin de mettre à jour ses listes.

À la notification de l'événement `evt_réclame_emplacement`, la politique choisit la première page répliquée et esclave dont le lien est primaire. Si la liste est vide ou que toutes les pages ont un lien secondaire, elle choisit de la même manière dans la liste des pages répliquées et dont la maîtrise est locale et, en dernier ressort, dans la listes des pages non répliquées.

La politique MRU fonctionne de la même manière sauf que les listes sont gérées de manière MRU. La politique RAND en revanche maintient également trois listes mais ne réagit pas à l'événement `evt_début_accès`. Lorsque l'événement `evt_réclame_emplacement`

lui est notifié elle choisit au hasard dans les listes en respectant toutefois la même priorité entre les listes.

Lorsqu'aucun des contextes locaux ne désire conserver un lien primaire, l'événement `evt_lien_détruit` est notifié par le système et récupéré par la politique de remplacement globale attachée au contexte M. S'il s'agit d'une page non répliquée, la politique consulte alors ses statistiques pour décider si la page doit être envoyée vers une autre machine et si oui, vers quelle machine elle doit être envoyée. Lorsque la page (ainsi que la maîtrise) est reçue par la machine destinataire, un lien est créé dans le contexte représentant le cache global. Afin d'en être avertie, la politique LRU qui est attachée à ce contexte est abonnée à l'événement `evt_lien_créé`.

7.4.2 Adaptation du serveur

Afin d'illustrer comment le support ADAMS permet de changer de politiques en cours d'exécution, nous montrons de quelle manière il est possible d'augmenter le serveur par l'introduction de garanties de type temps-réel. Bien entendu, cela ne peut être effectué de manière instantanée et sans effets. Ce qui nous semble important cependant est que ces changements affectent le moins possible l'exécution des transactions en cours et soit ainsi le plus transparent possible pour les utilisateurs du serveur.

Pour réaliser cela, le serveur doit pouvoir évoluer progressivement vers un serveur temps-réel et passant par un état intermédiaire entre serveur classique et serveur temps-réel. Les politiques remplacées et les politiques remplacantes doivent donc pouvoir coexister pendant un laps de temps, les premières responsables des transactions actives au commencement du basculement, les deuxièmes responsables des nouvelles transactions. Comme nous allons le voir, cela est rendu possible par le support ADAMS grâce à la dynamicité des contextes et des espaces de gestion.

Les politiques concernées par cette évolution sont les politiques d'allocation et de C&S. Dans un premier temps, les nouvelles versions sont chargées dans le système. celles-ci réagissent aux mêmes types d'événements que les anciennes mais de manière différente. La nouvelle politique d'allocation est identique à la première sauf qu'elle tient compte des priorités[3] des transactions dans sa répartition des emplacements. Un minimum d'emplacements est toujours accordé à chacune des transactions mais les emplacements excédentaires ne sont plus répartis équitablement mais en tenant

[3]Nous supposons que le gestionnaire d'exécution des transactions est maintenant capable d'attribuer une priorité à chaque transaction et de le faire connaître au gestionaire mémoire lors de la création de la transaction. Encore une fois, le système Arias permet ce flot d'information de l'espace utilisateur vers le module de spécialisation.

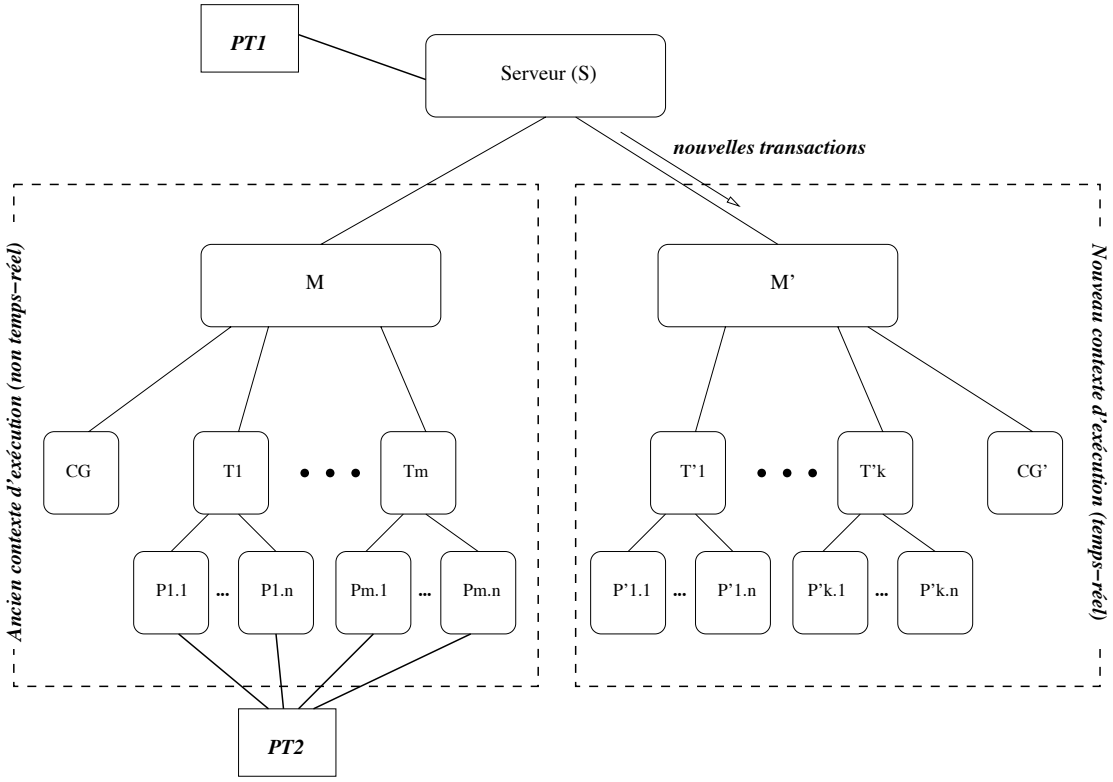


FIG. 7.10 – Basculement du serveur d'un mode d'exécution vers un autre.

compte de leur niveau de priorité : les emplacements sont en priorité attribués aux transactions les plus prioritaires. La nouvelle politique de C&S diffère de l'ancienne du fait de son comportement similaire à la politique 2PL-High-Priority. En effet, lors d'une inversion de priorité dû à un conflit d'accès entre deux transactions, la politique rejette la transaction la moins prioritaire afin de laisser passer celle plus prioritaire.

Si ces politiques étaient rattachées à des contextes créés à la volée (tels que les contextes T_i et $P_{i,j}$), il suffirait d'attacher les nouvelles versions aux nouveaux contextes créés. Cela n'est malheureusement pas le cas et explique l'existence du contexte M. En effet, comme le montre la figure 7.10, un nouveau contexte M', jumeau du contexte M, est créé avec un espace de gestion initialement nul. Les nouvelles politiques sont rattachées à ce nouveau contexte tandis que les anciennes restent attachées au contexte M. Lors de la notification des événements du type `evt_pas_de_ctx_terminal`, le module de spécialisation n'agrandit plus l'espace de gestion du contexte M mais celui du contexte M'. En d'autres termes, toute nouvelle transaction est "redirigée" vers le nouveau contexte d'exécution temps-réel tandis que les anciennes transactions continuent leur exécution dans le contexte non temps-réel.

Se pose maintenant le problème des deux contextes d'exécution. Concernant l'al-

location, il convient de répartir les emplacements entre les contextes M et M'. La valeur maximale d'allocation du contexte M est réduite de manière à augmenter celle du contexte M'. Cette réduction est effectuée progressivement et proportionellement à la différence du nombre de transactions s'exécutant dans l'ancien mode et celles s'exécutant dans le nouveau. Lorsque toutes les transactions s'exécutant dans l'ancien mode ont terminé leur exécution, le contexte M' a finalement récupéré la totalité de la mémoire. Notons que compte tenu de la répartition des emplacements entre les contextes T_i et le contexte CG, les transactions s'exécutant localement ne sont pénalisées que lorsque l'espace d'allocation du contexte CG est nul. Dit autrement, les transactions distantes sont d'abord pénalisées avant que ne le soient les transactions locales.

Le problème est cependant plus complexe pour la politique de C&S. En effet, l'ancienne et la nouvelle politiques ne partageant pas les mêmes structures de données, il n'est pas possible de gérer les conflits d'accès à une même donnée par des transactions ne s'exécutant pas dans le même mode. Afin de ne pas pénaliser les transactions s'exécutant dans l'ancien mode, toute nouvelle transaction tentant d'accéder une relation déjà accédée par une ancienne transaction est temporairement bloquée. Cela est réalisé par l'introduction de deux politiques temporaires PT_1 et PT_2. La première est attachée au contexte du module de spécialisation est récupère les événements du type `evt_début_accès`. En utilisant les fonctions d'introspection fournies par le support, elle vérifie que lors d'un accès, le segment correspondant à la relation accédée n'est pas rattaché à un contexte terminal appartenant à l'ancien mode. Dans le cas contraire, l'événement est tout simplement bloqué. La politique PT_2 est quant à elle rattachée à tous les contextes terminaux de l'ancien mode et s'abonne à l'événement `evt_contexte_détruit`. À la terminaison d'une transaction s'exécutant dans l'ancien mode, les contextes terminaux sont détruits par la politique d'allocation du contexte M et la politique PT_2 en est informée. À chaque destruction de contexte, celle-ci vérifie alors si cette destruction permet de débloquer une transaction de type temps-réel. Si cela est le cas, elle en informe la politique PT_1 afin qu'elle débloque l'événement `evt_début_accès` qui peut alors être notifié aux différentes politiques de gestion.

Lorsque plus aucune transaction ne s'exécute dans l'ancien mode, la politique PT_2 n'est alors plus rattachée à aucun contexte, la poltique PT_1 est détachée du contexte S et elles sont toutes deux déchargées du noyau. Le contexte M est également détruit et les anciennnes politiques sont déchargées du noyau. Ainsi, lorsque le basculement est terminé, le serveur s'exécute comme s'il avait toujours été un serveur de type temps-réel.

7.5 Conclusion

Nous avons présenté dans ce chapitre une implantation du support ADAMS dans le contexte du système Arias. Ce dernier offre un support pour la construction d'applications réparties sur une grappe de machines et nécessitant la permanence des données qu'elles manipulent. Il offre au niveau utilisateur une Mémoire Virtuelle Répartie et Partagée (MVRP) permanante et une architecture permettant d'ajouter et remplacer des modules de spécialisation de la gestion mémoire.

Intégré à ce système, ADAMS augmente ses capacités d'adaptation en tenant compte des besoins de gestion propres aux serveurs de données et en offrant une infrastructure permettant un grain d'adaptation plus fin qu'auparavant. La notification d'événements, mécanisme critique du mode d'interaction ouvert que nous proposons, se traduit par un simple appel de fonction et une faible occupation mémoire.

Afin de montrer la puissance de notre support, nous avons présenté comment il permet d'implanter aisément un noyau transactionnel réparti en intégrant, sans cependant créer de dépendances, des politiques de gestion relativement sophistiquées. Conformément au modèle d'organisation sur lequel repose ADAMS, le gestionnaire mémoire du noyau transactionnel se traduit par une hiérarchie de contextes auquels sont rattachées des modules de gestion appartenant aux différentes politiques implantées. Nous avons de plus montré comment ce modèle, en s'appuyant sur des concepts simples mais adaptés au problème de l'adaptabilité, permet de faire basculer le serveur en douceur vers une version offrant de nouvelles qualités de service. Grâce à lui, de nouvelles politiques peuvent être introduites puis intégrées dynamiquement au coeur du système tout en maximisant la continuité du service offert.

Chapitre 8

Conclusion

8.1 Bilan et contribution

Les serveurs de données, à la base des systèmes d'information, sont amenés à effectuer de plus en plus de traitements sur des volumes de données de plus en plus importants. On ne compte plus aujord'hui la varité des serveurs : systèmes de gestion de base de données, serveur Web, entrepôts de données, serveurs de couriel, serveurs de données scientifiques, serveurs multimédia, etc. Chacun chacun d'eux exécute des traitements dont les comportements sont eux-mêmes variés et bien souvent spécifiques au serveur. Également, les garanties qu'ils apportent quant à l'exécution de ces traitements (propriétées ACID, propriétés temps-réel, etc.) sont elles aussi bien spécifiques et nécessitent d'être impalntées judicieusement afin de ne pas compromettre les performences de ces serveurs. S'il on ajoute à cela la vitesse à laquelle, grâce à la crsoissance d'Internet, le monde informatique évolu, il devient impératif de construire des serveurs capable de s'adapter, d'évoluer en termes de puissance et de comportement. Les processus de conception et d'implantation doivent être aussi courts que possible. De plus, l'impact du processus d'adaptation sur la continuité des services offerts par le serveur doit être réduit au minimum.

Le point de départ de ce travail était l'étude de l'apport du système Arias pour la construction de serveurs de données en tant qu'applications réparties. Arias présente l'abstraction d'une mémoire virtuelle répartie partagée et persistante et constitue de ce fait une base très attractive pour la construction d'applications réparties sur une grappe de machines. Plutôt que de définir statiquement un ensemble de politiques de gestion, le système adopte une approche adaptable en permettant l'ajout dynamique de modules de spécialisation. Il offre également un ensemble de mécanismes qui doivent être offerts dans tout support mémoire : accès à l'espace de stockage, lecture

et écriture en mémoire distante, services de localisation et de communication, etc.

L'architecture d'Arias (à laquelle nous avons contribué) et notamment le découpage entre les mécanismes et les politiques ainsi que la possibilité d'ajouter et retirer dynamiquement des modules de spécialisation nous semble bien entendu très intéressante. Toutefois, Arias offrant une mémoire virtuelle répartie, sa conception avait essentiellement pour but de permettre aux utilisateurs de définir leurs propres politiques de cohérence et synchronisation. Le système convenait par conséquent bien pour des applications parallèles pour le calcul scientifique dont le principal soucis est le partage de données logiques entre tâches réparties. Il convenait en revanche moins pour la construction de serveurs de données qui sont amenés à manipuler de gros volumes d'information. Des fonctions pour la persistance et la résistance aux pannes étaient bien présentes dans le système mais sans soucis de l'échange de données entre l'espace disque et la mémoire physique. Arias n'offrait notamment rien pour la gestion de copies et le partage de l'espace physique.

En proposant ADAMS, notre intention est de combler ces lacunes. Cependant, offrir simplement des mécanismes internes supplémentaires pour la gestion des politiques d'allocation, remplacement et préchargement ne nous a pas semblé suffisant. En effet, cela aurait signifié que toutes les politiques de gestion d'un serveur auraient été regroupées dans un seul module de spécialisation. Changer une politique sans changer les autres n'aurait donc pas été possible et serait allé à l'encontre de la continuité de service. Bien qu'il existe des dépendances entre les différentes politiques, elles ont néanmoins chacune des responsabilités bien distinctes. En s'appuyant sur un modèle de gestion hiérarchique et sur un modèle de communication par événements, ADAMS permet au contraire de dissocier les différentes politiques de sorte qu'il est alors possible de changer une politique indépendemment des autres.

La première contribution de ce travail est un état de l'art des techniques de gestion d'une mémoire répartie. Nous avons présenté dans un premier temps les politiques de remplacement et de préchargement. Celles-ci ont comme objectif principal de limiter l'impact de la limitation de l'espace physique sur les performances du système en exploitant au mieux les ressources réparties disponibles. Dans un deuxième temps, nous avons présenté les politiques d'allocation et de cohérence et synchronisation (C&S) dont le rôle principal est d'isoler les traitements quant à l'utilisation des ressources physiques et logiques. Dans les deux cas nous avons montré la variété des politiques qui peuvent être trouvées dans la littérature.

Bien entendu, cette étude n'est pas, et loin s'en faut, exhaustive. Elle ne discute en particulier pas des problèmes de réplication et tolérance aux pannes et se concentre au contexte particulier des grappes de machines. Elle nous semble cependant une base

intéressante pour la réalisation d'une introduction compréhensive aux problèmes de gestion mémoire des serveurs de données répartis, introduction qui n'existe pas à l'heure actuelle. Elle montre de toute manière que d'une manière générale l'intérêt d'une politique dépend d'un grand nombre de facteurs et en particulier du comportement des traitements, des objectifs de performance visés et des qualités de service désirées. Il n'existe donc pas de politique meilleure que les autres ; chacune d'elles est adaptée à ce contexte particulier et doit pouvoir être utilisée lorsque cela est possible.

Nous avons vu ensuite l'apport d'un support mémoire en termes de réutilisation, factorisation et facilité d'évolution et l'intérêt de pouvoir l'adapter en fonction des besoins et comportement des traitements. Offrir un support qui peut être adapté dynamiquement est particulièrement important dans un contexte où la continuité de service doit être garantie. Parmi les systèmes existants nous avons pu constater que certains offrent un bon niveau de support mais ne sont malheureusement pas ou peu adaptables. D'autres en revanche offrent un degré d'adaptabilité (certes limité) mais ne fournissent pas suffisamment de mécanismes ou abstractions pour faciliter significativement la construction de serveurs de données répartis. Aucun ne permet à la fois l'ajout de politiques de remplacement, préchargement, allocation et cohérence et synchronisation spécifiques.

Nous avons en conséquence proposé le support mémoire ADAMS qui constitue la deuxième contribution de ce travail. ADAMS s'appuie sur un modèle de gestion hiérarchique de la mémoire basé sur la notion de contexte auquel peuvent être attachées des politiques de gestion. Le support offre un ensemble de mécanismes permettant d'agir sur l'état du système ainsi que des primitives permettant aux politiques de s'abonner à des événements correspondant à des changements de l'état du système. Ces événements sont notifiés aux politiques abonnées afin qu'elles puissent prendre des décisions et agir en retour sur l'état du système. Les types d'événements définis par le support offrent une base nécessaire pour implanter les politiques de gestion que nous avons introduites. ADAMS reste suffisamment ouvert en permettant la définition de nouveaux types d'événements et la manière dont ils doivent être notifiés.

Nous avons finalement présenté comment nous avons intégré ADAMS au système Arias. Nous avons de plus montré une implantation de la gestion des événements offrant un coût d'utilisation acceptable. Le résultat de l'intégration d'ADAMS dans le système Arias est par conséquent un support mémoire adaptable pour la construction de serveurs de données répartis.

8.2 Limitations et perspectives

Ce qu'il faut retenir avant tout de ce travail est l'idée qu'il est important de concevoir des systèmes en gardant à l'esprit qu'ils ont de fortes chances d'être amenés à évoluer, pour les raisons que nous avons évoquées ou pour d'autres raisons. Dans les années qui viennent, les systèmes vont être de plus en plus présents et de plus en plus intégrés à notre vie quotidienne. Du fait de la multitude et la variété des applications existantes et à venir, l'inter-opérabilité des applications (qui ne se connaissent pas à l'avance) semble aujourd'hui reconnue comme une nécessité évidente. De la même manière, nous pensons que les systèmes, et applications en général, vont devoir affronter des contextes d'exécution et d'utilisation de plus en plus variés et non déterminés à l'avance.

Nous avons parlé d'adaptabilité dans un contexte très particulier qui est celui des serveurs de données s'exécutant sur une grappe de machines. Même dans ce contexte restreint, nous avons entre autre ignoré les problèmes de *tolérance aux pannes* et *haute disponibilité*. Nous pouvons alors nous demander si notre proposition permet de définir et intégrer d'autres types de politiques. Bien qu'ADAMS offre la possibilité d'étendre les types d'événements et d'abonner n'importe quelle politique à n'importe quel contexte, le modèle de gestion hiérarchique et le modèle de communication par événements est-il réellement adapté pour d'autres types de politiques que celles que nous avons considérées ? La manière de hiérarchiser la gestion d'une politique correspond-elle toujours à la manière de hiérarchiser une autre politique ? Ne vaudrait-il pas mieux permettre différentes hiérarchies pour différentes politiques ? Comment notre proposition peut-elle être appliquée dans un contexte autre que celui des grappes de machines et celui des serveurs de données ?

Un autre point que nous avons laissé à l'écart est le problème de la cohésion des différentes politiques. Même si nous supposons que chaque politique a des responsabilité différentes, comment s'assurer que leurs décisions sont toujours compatibles ? Et comment s'en assurer dans un contexte où une politique peut être remplacée à tout moment par une autre ? Est-il possible de définir une sorte de *modèle de compatibilité* de politiques à partir de leurs rôles respectifs et de leur dépendances ?

Dans le contexte qui nous intéressait, nous n'avons pas considéré comme importants les problèmes de sécurité liés à l'adaptabilité. Cela se justifiait par le fait que la personne qui introduit de nouvelles politiques est a priori celle chargée de l'administration du serveur et possède des privilèges (dans notre cas cette personne doit connaître le mot de passe de l'administrateur du système Aix). Il s'agit d'une personne de confiance qui est sensée connaître le comportement des modules qu'elle ajoute au

noyau. Bien que cela limite les chances d'introduire des programmes malveillants, elle ne limite en rien les chances d'introduire des programmes mal construits. En l'occurrence, une grande partie du temps d'implantation du système Arias et des modules de spécialisation a été consacrée à élucider des problèmes de programmation dans un contexte noyau, multi-programmé et réparti. En effet, tous les modules chargés dans le noyau partagent le même espace d'adressage et peuvent modifier par erreur des structures de gestion qui ne leur appartiennent pas. Nous pensons que de nombreux problèmes auraient pu être évités si nous avions pu utiliser un langage plus sûr que le langage C et des techniques d'isolation de code. De toute manière, si l'adaptabilité doit être considérée dans d'autres contextes que le nôtre, les *problèmes de sécurité* devront certainement être abordés.

Bibliographie

[ABB+86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tavanian, and M. Young. Mach : A new kernel foundation for unix development. In *USENIX 1986 Summer Conference Proceedings*, July 1986.

[Ald98] Saud A. Aldarmi. Real-time database systems : Concepts and design. Technical report, Departement of Computer Science, University of York, April 1998.

[Bar98] Gretta Bartels. Markov prediction for adaptive network memory prefetching. Senior Thesis, June 1998.

[BCE+94] B. Bershad, C. Chambers, S Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E.G. Sirer. SPIN - an extensible microkernel for application-specific operating systems services. Technical Report TR-94-03-03, University of Washington, Seattle, WA, 1994.

[BCF+97] Jérome Besancenot, Michèle Cart, Jean Ferrié, rachid Gerraoui, Philippe Pucheral, and Bruno Traverson. *Les Systèmes Transactionnels : concepts, normes et produits*. HERMES, 1997.

[BCF+99] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions : Evidence and implications. In *Proceedings of the INFOCOM '99 conference*, March 1999.

[BCL93] K. P. Brown, M. J. Carey, and M. Livny. Managing memory to meet multiclass workload response time goals. In *Proceeding of the 19th International Conference on Very Large Data Bases (VLDB'93)*, pages 328–341, Dublin, Ireland, August 24-27 1993. Morgan Kaufmann Publishers.

[BCL96] Kurt P. Brown, Michael J. Carey, and Miron Livny. Goal-oriented buffer management revisited. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2), 1996.

[BCZ91] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin : Distributed shared memory using multi-protocol release consistency. In A. I. Karsh-

mer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, number 563 in Lecture Notes in Computer Science, pages 56–60. Springer-Verlag, July 1991.

[Bel66] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM System Journal*, 5(2), 1966.

[BGG+91] A. Bricker, M. Gien, M Guillemont, J. Lipkis, D. Orr, and M. Rozier. A new-look at microkernel-based unix operating systems : Lessons in performance and compatibility. Technical Report TR-91-7, Chorus Systems, 1991.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, 1987.

[BKV98] Luc Bouganim, Olga Kapitskaia, and Patrick Valduriez. Memory-adaptive scheduling for large query execution. In *Proceeding of the 7th International Conference on Information and Knowledge Management (CIKM'98)*, pages 105–115, Bethesda, Maryland, USA, November 3-7 1998. ACM Press.

[BNS69] L. Belady, R. Nelson, and G. Shedler. An anomaly in the space-time characteristics of certain programs running in paging machines. *Communications of the ACM*, 12(6), june 1969.

[BP95] A. Biliris and E. Panagos. Transactions in the client-server EOS object store. In *Proceedings of the 11th International Conference on Data Engineering (ICDE'95)*, pages 308–315, Taipei, Taiwan, March 1995. IEEE Computer Society Press.

[CD85] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceeding of the 11th International Conference on Very Large Data Bases (VLDB'85)*, pages 127–141, Stockholm, Sweden, August 21-23 1985. Morgan Kaufmann.

[CDF+94] M. J. Carey, D. J. Dewitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2) :383–394, June 1994.

[CFKL95] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 188–197, New York, NY, USA, May 1995. ACM Press.

[CFL94] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994.

[CG94] Richard L. Cole and Goetz Graefe. Optimization of dynamic query evaluation plans. In *Proceeding of the 20th International Conference on Management of Data (SIGMOD'94)*, pages 150–160, Minneapolis, Minnesota, May 24-27 1994. ACM Press.

[CM86] Michael J. Carey and Waleed A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems*, 4(4) :338–378, November 1986.

[CR93] C. M. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. In *Proceeding of the 19th International Conference on Very Large Data Bases (VLDB'93)*, pages 342–353, Dublin, Ireland, August 24-27 1993. Morgan Kaufmann.

[CY89] Douglas W. Cornell and Philip S. Yu. Integration of buffer management and query optimization in relational database environment. In *Proceeding of the 15th International Conference on Very Large Data Bases (VLDB'89)*, pages 247–255, Amsterdam, The Netherlands, August 22-25 1989. Morgan Kaufmann.

[DG94] D. L. Davison and G. Graefe. Memory-contention responsive hash joins. In *Proceeding of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 379–390, Santiago de Chile, Chile, September 12-15 1994. Morgan Kaufmann.

[DG95] Diane L. Davison and Goetz Graefe. Dynamic resource brokering for multi-user query execution. In *Proceeding of the 21st International Conference on Management of Data (SIGMOD'95)*, pages 281–292, San Jose, California, May 22-25 1995. ACM Press.

[DS94] A. Dan and D. Sitaram. Buffer management policy for an on-demand video server. Technical report, IBM Corporation, T. J. Watson Research Center, P.O. Box 218, Yorktown Heights, 1994.

[DWAP94] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching : Using remote client memory to improve file system performance. Technical Report CSD-94-844, University of California, Berkeley, December 1994.

[EH84] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *Transactions on Database Systems (TODS)*, 9(4) :560–595, 1984.

[FCL92] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global memory management in client-server database architectures. In Li-Yan Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 596–609. Morgan Kaufmann, 1992.

[FMP+95] 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 (SOSP-15)*, pages 201–212, December 1995.

[FNS95] Christos Faloutsos, Raymond T. Ng, and Timos K. Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Transactions on Computers*, 44(4) :546–560, 1995.

[Fra96] M.J. Franklin. *Client data caching : A foundation for High-Performance Object Database Systems*. Kluwer Academic Publishers, 1996.

[GA94] Jim Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the Usenix Summer 1994 Technical Conference*, pages 197–208, Boston, MA, USA, June 1994. Usenix Association.

[GC97] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. In *Proceeding of the 17rd International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'97)*, volume 25(1) of *Performance Evaluation Review*, pages 115–126, Seattle, Washington, USA, June 15-18 1997. ACM Press.

[GH74] R. P. Goldberg and R. Hassinger. The double paging anomaly. In *AFIPS National Computer Conference*, pages 195–199, 1974.

[GR93] J. N. Gray and A. Reuter. *Transaction Processing : concepts and techniques*. Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA, 1993.

[GUW99] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Integration*. Prentice Hall, 1999.

[HS90] J. Huang and J. A. Stankovic. Buffer management in real-time databases. Technical Report TR90-65, University of massachusetts, 1990.

[IC91] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *Proceeding of the 17th International Conference on Management of Data (SIGMOD'91)*, pages 268–277, Denver, Colorado, May 29-31 1991. ACM Press.

[IOfS92] International Organization for Standardisation. Information processing systems - database language sql. Technical report, ISO/IEC 9075, 1992.

[JS94] Theodore Johnson and Dennis Shasha. 2q : A low overhead high performance buffer management replacement algorithm. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceeding of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 439–450, Santiago de Chile, Chile, September 12-15 1994. Morgan Kaufmann.

[Kap80] J. Kaplan. Buffer management policies in a database system. Master's thesis, U.C. Berkeley, 1980.

[KK95] Alfons Kemper and Donald Kossmann. Adaptable pointer swizzling strategies in object bases : Design, realization, and quantitative analysis. *The VLDB Journal*, 4(3) :519–566, July 1995.

[KR81] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2) :213–226, June 1981.

[Lam79] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9) :690–691, September 1979.

[LCC94] Chao-Hsien Lee, Meng Chang Chen, and Ruei-Chuan Chang. HiPEC : High performance external virtual memory caching. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'94)*, pages 153–164, Berkeley, CA, USA, November 1994. USENIX Association.

[LCK+99] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sang Lyul Min, Yookun Cho, Chong Sang Kim, and Sam H. Noh. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceeding of the 19th International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'99)*, volume 27(1) of *Performance Evaluation Review*, pages 134–143, Atlanta, Georgia, USA, May 1-4 1999. ACM Press.

[LWY93] Avraham Leff, Joel L. Wolf, and Philip S. Yu. Replication algorithms in a remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems*, 4(11) :1185–1204, November 1993.

[MA90] D. McNamee and K. Armstrong. Extending the mach external pager interface to accommodate user-level page replacement policies. In *Proceedings of the USENIX Association Mach Workshop*, 1990.

[MAM98] Cristina Duarte Murta, Virgilio A. F. Almeida, and Wagner Meira, Jr. Analyzing performance of partitioned caches for the WWW. In *Proceedings of the 3rd International WWW Caching Workshop*, June 1998.

[MD93] Manish Mehta and David J. DeWitt. Dynamic memory allocation for multiple-query workloads. In *Proceeding of the 19th International Conference on Very Large Data Bases (VLDB'93)*, Dublin, Ireland, August 24-27 1993. Morgan Kaufmann.

[MDK96] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 3–18, Berkeley, October 28–31 1996. USENIX Association.

[MKK95] Frank Moser, Achim Kraiss, and Wolfgang Klas. L/mrp : A buffer management strategy for interactive continuous data flows in a multimedia dbms. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proceeding of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 275–286, Zurich, Switzerland, September 11-15 1995. Morgan Kaufmann.

[NDD92] V. F. Nicola, A. Dan, and D. M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *Proceeding of the 12th International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'92)*, volume 20(1) of *Performance Evaluation Review*, page 35, Newport, Rhode Island, USA, June 1-5 1992. ACM Press.

[NFS91] Raymond T. Ng, Christos Faloutsos, and Timos K. Sellis. Flexible buffer allocation based on marginal gains. In *Proceeding of the 17th International Conference on Management of Data (SIGMOD'91)*, pages 387–396, Denver, Colorado, May 29-31 1991. ACM Press.

[NHM+98] Norifumi Nishikawa, Takafumi Hosokawa, Yasuhide Mori, Kenichi Yoshida, and Hiroshi Tsuji. Memory-based architecture for distributed

WWW caching proxy. *Computer Networks and ISDN Systems*, 30(1–7) :205–214, April 1998.

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceeding of the 19th International Conference on Management of Data (SIGMOD'93)*, pages 297–306, Washingtion, D.C., May 26-28 1993. ACM Press.

[OOW99] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. An optimality proof of the lru-k page replacement algorithm. *Journal of the ACM*, 46(1), 1999.

[OV99] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, second edition edition, 1999.

[ÖVU98] M. Tamer Özsu, Kaladhar Voruganti, and Ronald C. Unrau. An asynchronous avoidance-based cache consistency algorithm for client caching DBMSs. In *Proceeding of the 24th International Conference on Very Large Data Bases (VLDB'98)*, pages 440–451, New York City, USA, August 24-27 1998.

[PCL93] HweeHwa Pang, Michael J. Carey, and Miron Livny. Partially preemptive hash joins. In *Proceeding of the 19th International Conference on Management of Data (SIGMOD'93)*, pages 59–68, Washingtion, D.C., May 26-28 1993. ACM Press.

[PCL94] HweeHwa Pang, M. J. Carey, and M. Livny. Managing memory for real-time queries. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2) :221–232, June 1994.

[PGG+95] R. Hugo Patterson, Garth A. Gibson, Eka Gintin, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the 15th Symposium of Operating Systems Principles*, pages 79–95, 1995.

[PZ91] Mark Palmer and Stanley B. Zdonik. Fido : A cache that learns to fetch. In *Proceeding of the 17th International Conference on Very Large Data Bases (VLDB'91)*, pages 255–264, Barcelona, Catalonia, Spain, September 3-6 1991.

[Rah91] Erhard Rahm. Concurrency and coherency control in database sharing systems. Technical report, Kaiserslautern, 1991.

[RD89] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. Technical report, 981, IBM

Research Division. T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598, 1989.

[Rei76] Allen Reiter. A study of buffer management policies for data management systems. Technical report, Mathematics Research Center, U. of Wisconsin-Madison, March 1976.

[RMSW97] Dan Revel, Dylan McNamee, David Steere, and Jonathan Walpole. Adaptive prefetching for device independent file I/O. Technical Report CSE-97-005, Oregon Graduate Institute of Science and Technology, July 18, 1997.

[RV98] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. Technical Report RN/98/13, UCL-CS, 1998.

[Sai97] Yasushi Saito. Transaction support in the spin operating system. Technical report, Department of Computer Science and Engineering, University of Washington, 1997.

[SE93] J. Srivastava and G. Elsesser. Optimizing multi-join queries in parallel relational databases. In *Parallel and Distributed Information Systems (PDIS '93)*, pages 84–92, Los Alamitos, Ca., USA, January 1993. IEEE Computer Society Press.

[SFPB95] G. Sirer, M. Fiuczynski, P. Pardyak, and B. N. Bershad. Safe dynamic linking in an extensible operating system. Technical Report TR-95-11-01, University of Washington, Department of Computer Science and Engineering, November 1995.

[SKW99] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU : Simple and effective adaptive page replacement. In *Proceeding of the 19th International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'99)*, volume 27(1) of *Performance Evaluation Review*, pages 122–133, Atlanta, Georgia, USA, May 1-4 1999. ACM Press.

[Smi78] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 3(3) :7–21, dec 1978.

[SPB88] A. Z. Spector, R. F. Pausch, and G. Bruell. Camelot : A flexible, distributed transaction processing system. *Spring COMPCON 88*, pages 432–437, March 1988.

[Spi76] Jeffrey R. Spirn. Distance string models for program behavior. *Computer*, 9(11) :14–20, November 1976.

[SS82] G.M. Sacco and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceeding of the 8th International Conference on Very Large Data Bases (VLDB'82)*, Mexico City, Mexico, September 8-10 1982.

[SS86] G.M. Sacco and M. Schkolnick. Buffer management in relational database systems. *Transactions on Database Systems*, 11(4), Dec 1986.

[Sto81] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7) :412–418, July 1981. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan Kaufmann, San Mateo, CA, 1988.

[SW97] M. Sinnwell and G. Weikum. A cost-model-based online method for distributed caching. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 532–542, Washington - Brussels - Tokyo, April 1997. IEEE.

[SZ90] Eugene J. Shekita and Michael J. Zwilling. Cricket : A mapped, persistent object store. Technical Report CS-TR-1990-956, University of Wisconsin, Madison, August 1990.

[TD91] C. D. Tait and D. Duchamp. Detection and exploitation of file working sets. In *11th International Conference on Distributed Computing Systems*, pages 2–9, Washington, D.C., USA, May 1991. IEEE Computer Society Press.

[TPH] The trapeze project home page. Duke University, Systems and Architecture, http ://www.cs.duke.edu/ari/trapeze/.

[TTL98] Anthony K. H. Tung, Y. C. Tay, and Hongjun Lu. BROOM : Buffer replacement using online optimization by mining. In *Proceeding of the 7th International Conference on Information and Knowledge Management (CIKM'98)*, pages 185–192, Bethesda, Maryland, USA, November 3-7 1998. ACM Press.

[VLN95] S. Venkataraman, M. Livny, and J. F. Naughton. The impact of data placement on memory management for multi-server OODBMS. In P. S. Yu and A. L. P. Chen, editors, *Proceedings of the 11th International Conference on Data Engineering*, pages 355–364, Los Alamitos, CA, USA, March 1995. IEEE Computer Society Press.

[VNL98] Shivakumar Venkataraman, Jeffrey F. Naughton, and Miron Livny. Remote load-sensitive caching for multi-server database systems. In *Pro-*

ceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA, pages 514–521. IEEE Computer Society, 1998.

[Whi94] Seth J. White. Pointer swizzling techniques for object-oriented database systems. Technical Report CS-TR-1994-1242, University of Wisconsin, Madison, September 1994.

[WR91] Y. Wang and L. A. Rowe. Cache consistency and concurrency control in a client/server DBMS architecture. In *Proceeding of the 17th International Conference on Management of Data (SIGMOD'91)*, page 367, Denver, Colorado, May 29-31 1991. ACM Press.

[WZ86] H. Wedekind and G. Zoerntlein. Prefetching in realtime database applications. In *ACM SIGMOD*, 1986.

[Yao77] S.B. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4) :260–261, Apr 1977.

[YC93] P. S. Yu and D. W. Cornell. Buffer management based on return on consumption in a multi-query environment. *The VLDB Journal*, 2(1) :1–38, January 1993.

[ZG90] H. Zeller and J. Gray. An adaptive hash join algorithm for multiuser environments. In Dennis McLeod, Ron Sacks-Davis, and Hans Schek, editors, *Proceeding of the 16th International Conference on Very Large Data Bases (VLDB'90)*, page 186, Brisbane, Queensland, Australia, August 13-16 1990. Morgan Kaufmann.

[Zip49] G. K. Zipf. *Human Behavior and the Principle of Least Effort : An Introduction to Human Ecology.* Addison-Wesley, Reading, MA, 1949.

[ZL97] Weiye Zhang and Per-Åke Larson. Dynamic memory adjustment for external mergesort. In *Proceeding of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 376–385, Athens, Greece, August 25-29 1997.