Conception et réalisation d'un service de stockage fiable et extensible pour un système réparti à objets persistants

Alain Knaff

THÈSE

présentée par

Alain Knaff

pour obtenir le titre de

Docteur de l'Université Joseph Fourier – Grenoble I

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Spécialité : Informatique

Conception et réalisation d'un service de stockage fiable et extensible pour un système réparti à objets persistants

Date de soutenance : 21 octobre 1996

Composition du jury :

Président :	Roland Balter
Rapporteurs :	Georges Gardarin Jean-Marie Rifflet
Examinateur :	Xavier Rousset de Pina
Directeur de thèse :	Sacha Krakowiak

Thèse préparée au Laboratoire Bull-IMAG et à l'INRIA–Rhône-Alpes

Remerciements

Je tiens à remercier :
Monsieur Roland Balter, Professeur à l'Université Joseph Fourier de Grenoble et responsable du projet Sirac, qui m'a fait l'honneur de présider le jury de cette thèse.
Monsieur Georges Gardarin, Professeur à l'Université de Versailles Saint-Quentin et Directeur du Laboratoire PRiSM, et Monsieur Jean-Marie Rifflet, Professeur à l'Université Denis Diderot (Paris VII), qui ont accepté d'être les rapporteurs pour mon travail,
Monsieur Sacha Krakowiak, Professeur à l'Université Joseph Fourier (Grenoble), qui m'a accepté au sein du Laboratoire Bull Imag, qui m'a encadré pendant mon travail, et dont les critiques furent toujours constructives.
Monsieur Xavier Rousset de Pina, Professeur à l'Institut Polytechnique (Grenoble), pour avoir accepté de participer au jury, et de porter sur mon travail un regard critique et enrichissant.
Ibaa Oueichek et Christian Jensen, mes voisins de bureau, pour l'ambiance superbe et les encouragements mutuels.
Je tiens également à mentionner tous les gens qui m'ont soutenu et encouragé pendant ces trois années de labeur, et plus particulièrement Christian, Daniel, Élizabeth, Fred, Jacques, Jay, Pascal, Pitch et Thierry pour les discussions fructueuses que nous avons eues durant tout ce temps, sans oublier tous les membres de l'Unité Mixte Bull-Imag et des projets Sirac et Dyade.

1

Introduction

L'avènement des processeurs rapides à 64 bits d'adresse [MSSW94, Dig92, 1st91] et des réseaux rapides [Par93] ont donné un nouvel essor au concept de mémoire virtuelle répartie [CLBHL92]. En effet, la disponibilité d'espaces d'adressage aussi grands encourage l'usage d'un modèle à *mémoire uniforme (Single Level Store)* [BTR78, Org72, Sol95, Wil91]. Ce modèle améliore essentiellement le support de deux composants clefs des systèmes d'information : la gestion de la mémoire et la désignation. En effet, la mémoire uniforme associe une adresse unique et universelle à chaque objet, qui permet donc de le désigner de manière non ambiguë.

Si nous étendons ce modèle à un environnement réparti, cela implique que l'on peut accéder aux données appartenant à la mémoire partagée simultanément à partir de plusieurs machines. Les données sont alors dupliquées dans la mémoire locale de chacune de ces machines.

Dans notre modèle, les données sont organisées en segments (unités logiques). Ces segments sont persistants, c'est-à-dire qu'ils survivent à la fin de l'application qui les a produits. Ils peuvent donc être réutilisés par d'autres applications.

Les segments doivent être rendus permanents (stockés sur disque) afin de les rendre résistant à une panne de système. Les systèmes d'information demandent souvent la capacité de pouvoir mettre à jour de manière atomique leurs données persistantes. Tout comme pour la cohérence de la mémoire vive, toutes les applications ne demandent pas le même niveau de garantie en ce qui concerne la cohérence entre les mises à jour et l'image permanente du segment. Nous proposons une solution souple, qui fournit une boîte à outils pour mettre en œuvre des mécanismes de mise à jour atomiques qui correspondent le mieux aux besoins des applications. Par conséquent, seules les applications qui ont vraiment besoin de l'atomicité doivent en payer le prix. Pour assurer l'atomicité des mises à jour, nous utilisons un *journal*. Le journal est un espace de stockage séquentiel où sont stockés, au fur et à mesure de l'exécution des applications, des enregistrements décrivant les modifications effectuées par ces applications sur la base. Notre journal contient des enregistrements ayant un format défini par l'application.

Aussi, nous fournissons aux applications les fonctions de base pour construire des systèmes personnalisés et résistant aux pannes. En faisant ceci, nous généralisons une idée de Gray. Gray reconnaît le besoin d'offrir aux applications un accès direct en lecture au journal, afin de réaliser diverses statistiques et optimisations [GR93].

Nous allons un pas plus loin, et nous permettons aux applications de décider quelles données elles souhaitent journaliser (le contrôle des écritures). Ainsi, nous tenons compte du fait que la nature des opérations à faire de manière atomique dépend fortement de l'application.

Afin de rendre notre système plus efficace, nous intégrons la gestion du stockage et la gestion de la mémoire virtuelle comme c'est fait dans Exodus [FZT+92a] et dans Quickstore [WD94]. Nous avons donc une vision globale de notre système de journalisation, qui interagit avec les autres composants du système. Sur ce point, notre travail est différent de Kitlog [Ruf92b]. Le but de Kitlog est de fournir une boîte à outils de journalisation, qui est certes plus souple que la nôtre, mais qui est moins intégrée dans un contexte global. L'intégration de notre système permet d'obtenir une meilleure efficacité à tous les niveaux : haut débit transactionnel et faible latence de validation.

1.1 Cadre de travail

Ce travail a été réalisé dans le cadre du projet Sirac (IMAG et INRIA). Le projet Sirac s'articule autour des deux axes suivants :

- La construction des **applications réparties**. L'objectif est de fournir des *outils* répondant à deux besoins :
  - La construction d'applications réparties en combinant des techniques de programmation à base d'objets et des techniques d'intégration de composants,
  - La simplification de l'administration, de la configuration et de l'évolution de ces applications.
- La réalisation de **services pour le support d'objets partagés persistants répartis**. L'objectif est de fournir un *support générique et efficace* utilisable pour la construction de plates-formes à objets répartis et de serveurs d'objets. Cette recherche s'appuie sur une réalisation prototype, Arias, qui réalise un service de mémoire virtuelle partagée répartie sur une grappe de machines. Sa conception tient compte des nouvelles infrastructures matérielles (grandes mémoires virtuelles et réseaux rapides). Le travail décrit dans cette thèse a été réalisé dans ce cadre.

La classe d'applications considérée en priorité est celle des applications coopératives, ce terme étant pris au sens large et couvrant notamment :

- La coopération autour d'une tâche commune d'équipes géographiquement réparties, avec partage étroit d'information (éventuellement multimédia) et interaction en temps réel. Exemples : téléconférence, édition coopérative de documents, aide à la décision, gestion coordonnée de ressources, aide à la formation.
- La coopération d'un ensemble d'outils partageant des données communes et devant se coordonner mutuellement. Exemples : systèmes de conception et fabrication assistées par ordinateur, ateliers de génie logiciel.

Le projet Sirac fait suite au projet Guide. L'expérience du projet Guide nous a permis d'identifier des problèmes pertinents, qui sont abordés dans Sirac.

1.2 Démarche

1.2.1 Objectifs et contraintes

L'objectif de ce travail est d'assurer le stockage permanent des segments de données. Ce service de stockage s'intègre dans un système de mémoire virtuelle répartie. Dans notre système, aussi bien la mémoire d'exécution que la mémoire physique de stockage sont réparties. Notre ambition est de faire un système performant : nous voulons assurer un débit de transactions élevé, et une faible latence de validation. Enfin, nous voulons offrir aux applications un service de stockage générique qu'elles peuvent adapter à leurs besoins.

L'opération de sauvegarde doit vérifier un certain nombre de propriétés qui seront détaillées par la suite (ACID).

1.2.2 Méthode proposée

Pour réaliser notre service de stockage, nous avons séparé la gestion des volumes (stockage à long terme) de la gestion du journal. Le gestionnaire de volume permet de stocker des images de segments sur disque. Il fournit des opérations de création et de destruction d'une image permanente d'un segment et de copie atomique d'une page d'un segment sur son image permanente. Le service de journalisation générique permet à l'utilisateur de créer un journal en lui associant un protocole spécifique de mise à jour et de reprise après panne. Ce service permet d'implanter indifféremment des journaux contenant des images avant, des images après, ou un mélange des deux.

Pour obtenir nos performances, nous intégrons notre système de stockage avec la gestion de la mémoire virtuelle. Nous proposons aussi des optimisations au niveau du stockage du journal physique.

1.2.3 Transactions dans une mémoire virtuelle répartie

Les systèmes transactionnels possèdent les propriétés ACID : atomicité, cohérence, isolation et durabilité [GR93]. Le terme *atomicité* désigne la propriété que toute opération qui se déroule dans le cadre d'une transaction forme une unité de travail unique et indivisible. Soit la transaction réussit entièrement, soit elle échoue entièrement. Il ne peut pas arriver qu'une transaction soit faite à moitié, même s'il y a des pannes. La cohérence est la propriété que toute transaction correcte qui part d'un état initial cohérent aboutit à un état final cohérent. Un état cohérent est un état qui vérifie les contraintes d'intégrité. La propriété d'*isolation* stipule que des opérations réalisées par des transactions différentes n'interfèrent pas entre elles, tant que les transactions en question ne sont pas terminées. Des modifications faites par une transaction qui n'a pas encore été validée ne sont donc pas visibles aux autres transactions. La *durabilité* est la garantie que des changements introduits par une

transaction validée sont acquis de manière durable et irréversible (sauf bien sûr si une transaction défait explicitement ces modifications).

Dans ARIAS, nous visons à intégrer le moteur transactionnel avec la mémoire virtuelle. Les applications transactionnelles travaillent directement dans la mémoire virtuelle répartie. À un moment donné, quand elles jugent que leur espace de travail est cohérent, elles *valident* la transaction. Ceci provoque, au niveau du système, une recopie de l'espace de travail concerné vers un endroit où il peut être retrouvé après la panne. La recopie se fait en deux temps :

- Au fur et à mesure que la transaction progresse, des enregistrements décrivant les modifications faites par l'application sont écrits dans le journal. Ceci se passe déjà avant la validation.

- Après la validation, le système *répercute* les modifications décrites dans le journal sur le support de stockage stable. Cette opération n'est pas urgente, étant donné que les opérations sont déjà enregistrées dans le journal. Nous pouvons donc nous permettre de la réaliser à un moment qui convient le mieux au système d'exploitation. C'est de cette réflexion qu'est née l'idée de coupler le mécanisme de répercussion des modifications sur l'image stable avec le mécanisme de pagination.

Le service de stockage se décompose donc en trois modules :

- Le *Service Générique de Journalisation*, qui est responsable de la journalisation bas niveau,

- Le *Gestionnaire de Volumes* qui est responsable du stockage de l'image stable,

- Le *Protocole Spécifique de Journalisation* qui offre aux applications une vision personnalisée du système.

1.3 Plan de la suite

Le reste de cette thèse est organisé comme suit :

Chapitre 2 Cet état de l'art est articulé autour de trois axes. D'abord nous traitons l'aspect interface des systèmes de stockage : on peut traiter les données persistantes comme des fichiers, ou comme des objets persistants ou comme de la mémoire uniforme. Ensuite nous examinerons comment les systèmes de stockage mettent en œuvre la fiabilité. Finalement, nous montrons comment ils intègrent les aspects gestion de la mémoire virtuelle avec la gestion de stockage.

Chapitre 3 Ce chapitre contient une description de l'ensemble du système Arias. Au cours de cette description, nous abordons d'abord ses objectifs et principes. Ensuite nous passons en revue brièvement les différents composants d'Arias. Nous décrivons en particulier le service de stockage et ses relations avec les autres composants.

Chapitre 4 Le quatrième chapitre se concentre sur la conception du service de stockage, et en donne les principes directeurs. Il illustre les problèmes que nous avons rencontrés et ébauche des solutions.

Chapitre 5 Ce chapitre donne des détails sur la mise en œuvre du service de stockage. Il en passe en revue ses différents composants, en précisant à chaque fois les problèmes techniques rencontrés et en détaillant les solutions adoptées.

Chapitre 6 Ce chapitre propose une évaluation des performances de notre système. Nous étudions notamment l'influence de la répartition et du degré de concurrence.

Chapitre 7 La conclusion résume les différents aspects abordés par cette thèse en mettant l'accent sur les aspects fonctionnels, leur mise en œuvre et les apports du travail effectué. Elle se termine par les perspectives.

Appendice A Le premier appendice décrit les mécanismes de AIX que nous avons utilisés pour Arias, ainsi que les problèmes techniques que nous avons rencontrés. Notre mise en œuvre utilise les streams comme moyen de communication entre modules. Le modèle des streams ainsi que l'utilisation que nous en faisons sont présentés en détail.

Appendice B Le second appendice décrit les interfaces externes et internes des modules de stockage.

2

Stockage des données : État de l'art

Le but de cette thèse est de décrire la conception du service de stockage d'Arias, un système de mémoire virtuelle réparti à 64 bits d'adresse. Pour faire ceci, nous devons d'abord examiner l'état de l'art de la recherche en systèmes suivant deux aspects :

- L'interface offerte par les systèmes de stockage à leurs applications,
- La mise en œuvre des systèmes de stockage, en particulier de la résistance aux pannes.

Ce chapitre est donc subdivisé en quatre sections. La première passe en revue les trois interfaces possibles au stockage permanent, à savoir l'abstraction fichiers, les objets persistants, et la mémoire virtuelle. Nous nous attardons sur ces questions d'interfaces, car une des forces d'Arias est sa mémoire virtuelle partagée résistante aux pannes.

La seconde section illustre la mise en œuvre du stockage. Nous distinguons trois techniques : la première sont les points de reprise, qui visent à sauvegarder l'état d'exécution courant, et permettent ainsi de reprendre des calculs interrompus. Les deux autres techniques visent avant tout à sauvegarder un état *cohérent*, c'est-à-dire un état entre deux calculs, ou entre deux transactions. L'état de la transaction courante est perdu. Ces techniques sont les *versions ombres* et la journalisation.

Dans Arias, nous avons choisi d'offrir aux applications une interface de mémoire virtuelle partagée résistante aux pannes. Notre service de stockage utilise un journal. Nous illustrons donc dans la troisième section comment les différents systèmes intègrent cette interface avec cette mise en œuvre.

La quatrième section est une conclusion de ce chapitre.

2.1 Interface des systèmes de stockage

Dans cette section, nous allons exposer la vision du stockage que les différents systèmes offrent à leurs applications.

Historiquement, l'abstraction la plus ancienne est celle *des fichiers*. Un fichier est simplement une suite linéaire d'octets. Chaque fichier est identifié par son *nom de fichier*, qui se décompose en son *chemin d'accès* et son *nom de base*. L'inconvénient des

fichiers est qu'ils ne permettent pas facilement de stocker des structures complexes. En effet, lors de la sauvegarde dans un fichier, toute structure doit être *linéarisée*, c'est-à-dire transformée en une suite linéaire d'octets sans pointeurs ni branchements. Un autre inconvénient de l'approche fichier est qu'elle nécessite une sauvegarde et un chargement explicites. L'application doit effectivement gérer deux espaces : l'espace d'exécution qui correspond à la mémoire vive (primaire) qui peut contenir des structures complexes, et l'espace de stockage, correspondant à la mémoire secondaire. Étant donné que le modèle « fichiers » est utilisé par tous les systèmes d'exploitation « grand public », nous considérons qu'il est suffisamment bien connu pour ne pas nécessiter d'exemples pour l'illustrer.

Une interface plus élaborée est constituée par les *objets persistants*. Cette approche permet de stocker directement les structures complexes dans l'espace persistant. De plus, l'accès aux deux espaces se fait d'une manière unique, ce qui efface la distinction entre espace d'exécution et espace de stockage et facilite ainsi la programmation des applications. Physiquement, les deux espaces continuent bien sûr d'exister, mais le système s'occupe à masquer la différence entre les deux, en gérant automatiquement la transition des objets du disque vers la mémoire, et vice versa. Les systèmes à objets persistants traditionnels sont basés sur des architectures à 32 bits. Cependant, une base d'objets contient souvent plus de données qu'il n'est possible d'adresser avec 32 bits d'adresse. Les adresses virtuelles ne peuvent donc pas être utilisées pour identifier un objet de manière univoque, et il faut une autre représentation, souvent appelée *OID* (Object Identifier). Les objets stockés sur disque sont liés entre eux par des OID, tandis que les objets se trouvant en mémoire vive utilisent comme pointeurs des adresses virtuelles. La traduction des références entre objets d'un format vers l'autre est réalisée de manière transparente par le système, lors du chargement et du déchargement des objets. Cette opération s'appelle le *swizzling*.

Malheureusement, le swizzling est une opération coûteuse. De plus, le swizzling devient inutilisable si à un moment donné la quantité des objets *actifs* (utilisés par un processus) est plus grande que l'espace virtuel. Heureusement, aujourd'hui, il existe des processeurs qui offrent un espace d'adressage de très grande taille (64 bits d'adresse), qui permettent donc de nommer les objets directement par leur adresse virtuelle, que ce soit en mémoire de stockage où en mémoire d'exécution. Il en résulte une grande simplification de la gestion de la base des objets : en effet aucune traduction n'est plus nécessaire. Cette abstraction est souvent appelée la *mémoire virtuelle répartie partagée*.

2.1.1 Exemples

Dans cette section, nous passerons en revue quelques systèmes mettant en œuvre les objets persistants ou la mémoire virtuelle répartie.

2.1.1.1 Monads

Le système `Monads` [RHB+90] a été développé à l'université de St. Andrews. Il utilise une grande mémoire virtuelle persistante dans laquelle les données persistent

jusqu'à leur destruction explicite plutôt que jusqu'au moment où le processus qui les a créées se termine. Ceci supprime tout besoin d'avoir un système de fichiers dans le sens habituel.

Le système `Monads` fournit un espace unique de noms persistant de 60 bits d'adresse. Cet espace est subdivisé en des espaces de 28 bits, dont chacun peut contenir des segments de code ou de données. L'accès à ces espaces d'adressage est accordé selon le paradigme orienté objet : les méthodes dans les segments de code d'un espace d'adressage sont le seul moyen d'accéder aux données de cet espace. Ces espaces, appelés *modules* sont protégés par des capacités qui définissent qui peut invoquer quelles méthodes dans ces modules.

Les capacités sont créées et protégées par le noyau, et il n'est pas possible à un programme utilisateur de détourner cette protection. De plus, `Monads` garantit qu'une ancienne capacité ne peut jamais être utilisée pour accéder à des données nouvelles. Ceci est réalisé en ne réutilisant jamais une zone de mémoire détruite. Bien que cela impose une limite supérieure à la quantité de données qui puisse jamais exister au sein du système, un million de Toctets est accepté comme étant une limite raisonnable.

Le système met aussi en œuvre une politique d'intégrité des données. Celle-ci garantit que lors d'une panne l'espace de stockage sera remis dans un état «correct». Ceci signifie que l'espace de stockage sera correct *de façon causale* après une panne, c'est-à-dire que si un effet est présent dans cet espace, la cause est présente elle aussi. Cette propriété est très importante, étant donné qu'un tel espace contient des références croisées implicites, et n'a pas de structure prédéfinie (à l'opposé d'un système de fichiers `Unix` qui a la structure d'un arbre). Donc toute erreur pourrait entraîner une brèche dans le système de capacités, ou une perte de données.

2.1.1.2 Clouds

Le système Clouds [DLAR91, DCM+91, BAHK+88] a été développé au Georgia Institute of Technology. Il met en œuvre une mémoire uniforme persistante à objets, semblable à celle présente dans `Monads`, et il est réalisé sur un réseau de stations de travail Sun 3. Trois types de machines existent dans un tel réseau :

- Des serveurs de calcul,
- Des serveurs de données,
- Des stations de travail utilisateurs,

`Clouds` utilise un modèle à objets passifs et à flots d'exécution actifs. Les objets passifs sont des structures de mémoire qui contiennent du code et des données qui ne disposent pas d'unité de calcul (flot d'exécution) associée. Les flots d'exécution actifs forment la ressource de calcul. Ces flots se déplacent d'objet en objet, y exécutent du code, et manipulent des données. Les objets portent des noms uniques, et sont composés d'un groupe de segments de mémoire. Chaque segment réside à une adresse différente au sein de l'objet et contient un type différent de données. Typiquement, un objet contient 4 segments :

- Du code persistant,

– Des données persistantes,

– Un tas (*heap*) volatile,

– Un tas persistant.

Le segment de code offre une interface procédurale à l'objet. Seul un flot d'exécution entrant dans l'objet en utilisant ses méthodes peut accéder aux données de l'objet. Les données sont déplacées entre objets uniquement en tant que paramètres d'appel et valeurs de retour de fonctions. Physiquement ces données passent par le segment pile associé au flot. Tous les segments au sein d'un objet sont partagés par tous les flots et cette pile fournit le seul espace de stockage indépendant d'un objet.

Chaque objet dispose de deux sous-espaces persistants. Toutes les données stockées dans ces sous-espaces persistent d'une invocation jusqu'à la prochaine. Les données volatiles ne sont pas préservées. Les données persistantes constituent la seule forme de stockage disponible dans `Clouds` et il n'y a donc aucun besoin de supporter un système de fichiers dans le sens classique. La souplesse des objets est améliorée par l'usage de techniques de mémoire partagée. Ceci permet à un même objet d'être invoqué sur des machines différentes, tout en gardant l'illusion d'un partage physique. Cette approche comporte beaucoup d'avantages pour la construction de grandes applications parallèles à base de mémoire partagée. Certes, il peut être plus avantageux dans certaines circonstances que tous les flots invoquent un objet donné sur la même machine physique. Ce mode de fonctionnement est également disponible.

Un objet `Clouds` fournit la *transparence du partage* en utilisant la mémoire virtuelle partagée répartie. Il fournit la *transparence de l'accès* en utilisant des noms uniques, et la *transparence de la localisation* en évitant toute association entre les noms et l'emplacement, une approche qui est simplifiée par l'interface procédurale obligatoire de l'objet. En plus, les objets fournissent à la fois du stockage persistant et du stockage volatile.

Couramment, aucune sécurité ou protection n'est mise en œuvre par le système `Clouds`. La capacité nécessaire pour invoquer un objet `Clouds` est accordée par le serveur de noms, et toute protection est mise en œuvre par le langage de programmation.

`Clouds` fournit un environnement de programmation parallèle souple. Il fournit un bon modèle de données partagées, qui est capable d'utiliser des machines réparties de manière transparente. La fiabilité, l'intégrité des données, et la tolérance aux pannes sont toutes gérées par l'usage de protocoles de validation de données, par le verrouillage de segments, et par des objets dupliqués et des flots. Aucune tentative n'a été faite pour gérer l'authentification où la protection au sein de `Clouds` autre que celle mise en œuvre par le langage. De plus, l'environnement ne peut être utilisé directement, mais on doit y accéder à travers un système `Unix`. L'administration répartie n'est pas traitée.

Cohérence des données Clouds fusionne le protocole de synchronisation (accès au structures de données partagées) avec la gestion de cohérence de la mémoire virtuelle répartie. Quand un processus demande un verrou sur une zone de données,

il ne récupère pas seulement le verrou, mais aussi les données qui y sont rattachés. Ceci se passe en une seule transaction réseau. Ce système a l'avantage que l'on n'a pas besoin de messages d'invalidation, l'obtention du verrou garantit qu'il n'y a pas d'autres copies en conflit avec celle qui est modifiée ; il a l'inconvénient qu'une opération explicite de déverrouillage est nécessaire pour libérer l'accès aux données. Ceci est acceptable dans Clouds : en effet le système DSM a été conçu spécialement pour supporter l'invocation d'objets, et il est donc facile de déclencher un déverrouillage implicite de l'objet sur un retour de méthode.

2.1.1.3 Opal

Opal [CLLBH92b, CLBHL92, CLLBH92a] est un système d'exploitation pour un environnement réparti, développé à l'Université de Washington, USA. Il définit un espace d'adressage unique qui donne accès à toutes les données du système, y compris aux données persistantes. L'espace global d'adressage est étendu à travers le réseau, afin de garantir l'unicité globale des adresses, et aussi afin de permettre la localisation des données grâce à leurs adresses virtuelles.

Les unités d'exécution sont les *flots*. Un *domaine de protection* fournit le contexte d'exécution pour les flots, en restreignant ainsi leur accès à un ensemble spécifique de segments à tout instant donné. Tous les domaines de protection partagent un même espace virtuel réparti. Les unités pour l'allocation de stockage et pour la protection sont les *segments*, qui sont des ensembles de pages contiguës. Les segments peuvent être persistants ou temporaires. L'adresse virtuelle d'un segment est un attribut permanent. L'adresse est choisie par le système au moment de l'allocation. Une fois créé, le segment peut être explicitement attaché à un domaine de protection, en permettant ainsi aux flots s'exécutant au sein de ces domaines d'accéder au segment. Les flots d'exécution, les domaines de protection, et les segments sont tous désignés par des *capacités*, qui contiennent des droits d'accès. Les domaines peuvent donc être considérés aussi bien comme ensembles de capacités (correspondant à l'ensemble des données accessibles par ce domaine) que comme ensemble de flots d'exécution qui se déroulent dans ce domaine. Les flots d'exécution se déroulant dans les différents domaines de protection peuvent communiquer entre eux en utilisant la mémoire partagée si les deux domaines, vus comme ensembles de droits, ont une intersection non vide. En plus, les flots d'exécution peuvent se déplacer entre les domaines. Un *portal* est un point d'entrée d'un domaine, identifié de manière non ambiguë par une valeur de 64 bits. Tout flot qui connaît la valeur du nom de portal peut faire un appel système qui transfère le contrôle vers le domaine de protection identifié par le portal.

Un prototype d'Opal est mis en œuvre au-dessus du micro-noyau Mach [CLFL93]. Les domaines de protection d'Opal sont mis en œuvre en tant que tâches Mach qui partagent toutes le même espace d'adressage. Les segments sont mis en œuvre comme des objets mémoire.

L'adressage à 64 bits d'adresse devient de plus en plus populaire : la même idée est reprise dans Mungi [HERV93] et Angel [MSWK93].

2.1.1.4 Texas

Texas [SKW92] est un système de mémoire virtuelle persistante orienté objets. Il fonctionne à base de swizzling : la représentation des pointeurs sur disque et en mémoire vive est différente. Ceci lui permet de fonctionner sur des machines à 32 bits d'adresse, et aussi de fusionner facilement plusieurs bases qui n'étaient pas liées entre elles auparavant. En effet, lors de la fusion de plusieurs bases, il peut arriver qu'une même adresse soit utilisée par deux objets différents dans les deux bases, si on suppose qu'il n'y a eu aucune coordination entre ces deux bases avant la fusion. Dans un système où les objets seraient nommés uniquement par leur adresse virtuelle, ceci poserait un problème presque insurmontable : Il faudrait relocaliser un des deux objets, et changer toutes les références qui y pointent. Dans Texas, cette situation ne pose pas de problème, car les objets sont nommés par des références logiques (OID) plutôt que matérielles. Les adresses virtuelles n'ont qu'une validité temporaire, donc la relocalisation ne pose pas de problème.

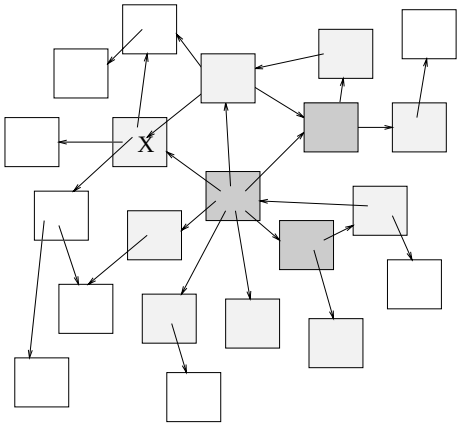

FIG. 2.1 – *État des pages avant chargement de la page X*

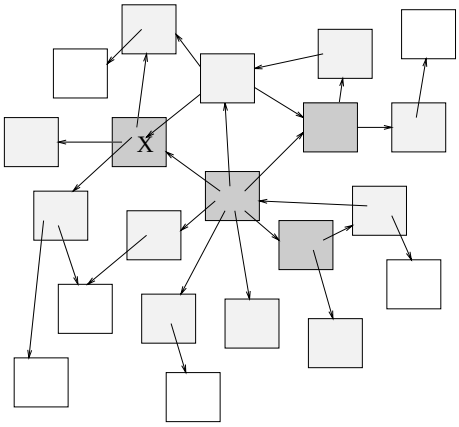

FIG. 2.2 – *État des pages après le chargement de la page X*

Le pointeurs contenus dans une page sont convertis lors du chargement de la page en mémoire vive. Si besoin est, l'espace virtuel pour les pages pointées par la page chargée est alloué immédiatement, mais ces nouvelles pages ne sont pas encore chargées. Ce chargement ne se fera que lorsque l'application y accédera directement. Ainsi, le chargement progressif des données peut se voir comme deux fronts d'onde qui se propagent l'un à la suite de l'autre dans la base : d'abord le front des pages allouées, mais non encore chargées, et ensuite le front des pages effectivement chargées en mémoire virtuelle. Les figures 2.1 et 2.2 illustrent ceci. Les pages noires sont chargées, les pages en pointillé sont allouées en mémoire virtuelle, mais pas chargées, et les pages blanches ne sont ni chargées, ni allouées. La figure 2.2 illustre ce qui se passe quand la page X est déréférencée par l'application.

Quand la mémoire virtuelle est remplie entièrement, toutes les pages qui y sont contenues sont éliminées[1], et le processus de construction du front d'onde commence à zéro. Ceci me semble très coûteux, mais d'après les auteurs un tel événement devrait être rare. Malheureusement, il est difficile de concevoir un algorithme qui

1. Elles sont protégées en accès. Elles peuvent cependant rester dans la mémoire physique.

soit plus économique. Il s'agit donc d'un problème de principe inhérent au modèle. En effet, il ne suffit pas d'éliminer simplement la page la moins récemment utilisée, car il pourrait y avoir des pointeurs sur cette page, et localiser ces pointeurs dans toute la mémoire virtuelle n'est pas une tâche évidente.

2.1.2 Synthèse

Monads, Clouds, Opal et Texas sont tous des projets de systèmes d'exploitation pour pour lesquels la gestion du stockage ne constitue qu'une partie. Un de leurs points commun est qu'ils fournissent tous une mémoire uniforme persistante, combinant mémoire d'exécution et mémoire de stockage en une seule ressource adressable directement. Les systèmes diffèrent par la manière dont est géré cet espace, et dont les objets sont nommés. Clouds, Monads et Texas utilisent des noms logiques pour les objets. Un même objet peut se trouver à des adresses virtuelles différentes selon l'instant et le contexte. Seul l'identificateur logique, c'est-à-dire l'OID, reste constant. Opal par contre utilise l'adresse virtuelle comme identificateur : les objets restent à des adresses virtuelles fixes qui ne bougent pas, et qui sont les mêmes dans tous les contextes.

La protection des données contre des accès non autorisés est réalisée de différentes manières dans ces systèmes. Monads et Opal utilisent de la protection matérielle à base de capacités, alors que la protection de Clouds repose entièrement sur le langage utilisé. Le problème de la protection n'est pas adressé dans Texas.

Clouds introduit la mémoire partagée réparti en tant que moyen de partager physiquement des données entre différentes machines. Un tel mécanisme est essentiel si on modélise un réseau de machines par une ressource unifiée. En absorbant l'accès vers des données distants dans un répositoire commun, un processus n'a pas besoin d'être conscient de l'origine des données qu'il manipule, ni de l'emplacement des autres processus avec lequel il partage ces données. Le problème du partage et de la persistance n'est plus le problème de l'application, mais celui du système.

La figure 2.3 présente une brève synthèse des systèmes présentés.

	Monads	Clouds	Opal	Texas
persistance	oui	oui	oui	oui
utilisation d'objets	passifs	passifs	non	passifs
accès aux données	grand espace	?	grand espace	swizzling
protection	capacités	mise en oeuvre par le langage	capacités	?
répartition	?	oui	oui	?

FIG. 2.3 – *Accès à la mémoire virtuelle*

2.2 Résistance aux pannes

La résistance aux pannes est trop souvent ignorée par les concepteurs de nouveaux systèmes d'exploitation, et rajoutée après coup quand le besoin s'en fait sentir. Au fur et à mesure que la taille des systèmes répartis augmente, leur fiabilité diminue, et le besoin de fournir de la résistance aux pannes devient plus important. Des essais pour rajouter de la résistance aux pannes après coup ont donné naissance à différentes approches, dont aucune ne peut résoudre tous les problèmes ou tenir compte de toutes les situations.

Dans cette section, nous examinons le support pour la résistance aux pannes dans les systèmes à mémoire virtuelle répartie.

Un système résistant aux pannes pour des applications générales doit être capable de :

- **Fournir de la résistance aux pannes sans modification des applications**, et permettre donc à n'importe quelle application d'utiliser ce service plutôt que de restreindre ce service à celles qui ont été spécialement conçues pour cela,
- **Fournir une solution complète plutôt que de supporter seulement une partie des services nécessaires**. En effet, une solution qui supporterait par exemple seulement de la mémoire résistante aux pannes, mais pas des fichiers serait inutile pour de nombreuses applications,
- **Supporter à la fois des applications mono- et multiprocesseurs**, et donc permettre de gérer correctement des applications parallèles sur des machines réparties,
- **Optimiser le cas général**, c'est-à-dire celui où il n'y a pas de problèmes. En effet, comme les pannes sont relativement rares, la protection contre les pannes ne doit pas avoir un coût prohibitif.

Les systèmes décrits dans cette section résolvent ces problèmes d'une manière ou d'une autre, mais aucune solution ne résoud tous les problèmes à la fois.

2.2.1 Sauvegarde et reprise

La première méthode pour assurer la résistance aux pannes que nous décrivons est le mécanisme de sauvegarde et reprise. Périodiquement, le système sauvegarde son état complet sur disque dans un *point de reprise*. Les données sauvegardées sont suffisamment complètes pour permettre une reprise des opérations en cours, à partir de l'état tel qu'il existait au moment de la sauvegarde. La prise de points de reprise sur les systèmes répartis est un problème non trivial, car il faut non seulement sauvegarder l'état de la mémoire utilisateur de chaque machine, mais aussi l'état des messages en transit. Le point de reprise doit être *cohérent*, c'est-à-dire correspondre à un état qui respecte la causalité : il ne faut pas par exemple qu'une partie des données contiennent les effets d'une opération, et d'autres ne contiennent pas la cause. La manière la plus simple d'assurer ceci est d'arrêter le système complètement lors de

la prise du point de sauvegarde. Cependant, cette solution n'est pas acceptable à cause du surcoût engendré. Donc en général le système est partitionné en plusieurs morceaux qui sont sauvegardables de manière indépendante, et qui peuvent donc aussi être arrêtés de manière indépendante, pendant que le reste du système continue à tourner.

Nous allons décrire deux systèmes, Monads et Clouds, qui utilisent des points de reprise pour assurer la fiabilité.

2.2.1.1 Monads

`Monads`[RHB+90] fournit une politique d'intégrité de données ; celle-ci est mise en œuvre par un mécanisme de point de reprise et de *rollback*.

Dans `Monads`, un module peut être sauvegardé à n'importe quel instant de son exécution. Quand une sauvegarde est demandée, le système protège les pages du module contre l'écriture, et met en œuvre une politique de *copy-on-write*. Cette opération de protection est rapide, et quand elle est terminée, le module peut continuer à s'exécuter. La sauvegarde proprement dite est faite de manière paresseuse en utilisant un protocole de validation à deux phases[2].

En intégrant ce mécanisme de fiabilité avec la mémoire virtuelle, `Monads` fournit un seul système de résistance aux pannes qui résout tous les problèmes d'intégrité des données. Cependant, cette solution ne traite pas les problèmes rencontrés dans les machines parallèles. En effet, la structure fermée des modules autorise seulement des échanges entre modules bien définis par des invocations de méthodes, et non les techniques plus générales basées sur la mémoire partagée utilisés dans la programmation parallèle. Ceci facilite la tâche d'établir les relations de dépendance de données entre les différents modules, mais complique l'écriture de programmes parallèles. Cette situation est simplifiée davantage comme il n'y a pas de moyen d'invoquer une méthode du même module sur plusieurs machines à la fois, donc pas de moyen de supporter du parallélisme basé sur les variables partagées.

2.2.1.2 Clouds

Comme déjà décrit précédemment, `Clouds` [DLAR91, DCM+91, BAHK+88] et `Monads` [RHB+90] offrent à leurs applications une représentation de données semblable. Ceci entraîne la nécessité d'éviter la corruption des données dans leur système de stockage persistant lors des pannes. Dans un système réparti, la *résilience* et la *fiabilité* sont des problèmes critiques. `Clouds` attaque ce problème de deux manières ; d'abord en fournissant un mécanisme de point de reprise semblable à celui de `Monads` et ensuite en fournissant des flots d'exécution dupliqués.

`Clouds` suppose qu'il n'est pas toujours nécessaire d'assurer l'intégrité d'un objet. Par exemple, les données temporaires n'ont pas besoin de survivre à une panne. `Clouds` fournit donc plusieurs niveaux de gestion d'intégrité des données. La politique désirée est associée au flot réalisant l'opération ; un *s-thread* ne fournit aucune garantie d'intégrité. Ce type de flot est le type par défaut. Un *cp-thread* maintient une cohérence complète de tous les objets modifiés.

2. Les protocoles de validation à deux phases seront discutés plus en détail dans la suite.

Quand un cp-thread accède à un segment au sein d'un objet, le segment est verrouillé pour la lecture et l'écriture selon le type d'opération réalisée. D'autres cp-threads sont alors empêchés d'accéder à ces segments. Quand le flot termine avec les segments verrouillés, les modifications sont écrites sur disque et validées en utilisant un protocole de validation à deux phases. Les segments sont alors déverrouillés, et rendus accessibles à tout autre cp-thread qui est éventuellement en attente. Pour une description plus complète, voir [CD89].

L'intégrité des données dans un tel système constitue seulement une partie du problème. Une résistance aux pannes complète est nécessaire afin de garantir qu'une panne de machine pendant l'exécution n'empêche pas la terminaison du programme. `Clouds` réalise ceci en utilisant des objets et des flots d'exécution dupliqués. Un mécanisme de validation atomique garantit que parmi ces flots dupliqués un seul résultat (correct) est produit. Tous les calculs sont réalisés en parallèle. Quand une instance du calcul se termine, elle est autorisée à valider ses changements. Si la validation marche, toutes les instances dupliquées des différents objets ayant servi au calcul, et tous les autres flots sont détruits. Si par contre la validation échoue, alors les objets utilisés par le flot qui a échoué sont écartés, et le système attend la complétion d'un autre flot. Le processus est répété jusqu'à ce que l'on ait le résultat, ou jusqu'à ce qu'il ne reste plus de flot. Avec un degré de réplication suffisant, et avec une probabilité d'erreurs suffisamment faible, le calcul se termine au bout du compte. Le détail de ce mécanisme est décrit dans [ADL90].

2.2.2 La journalisation

Une méthode très populaire pour assurer l'atomicité des transactions est le journal. Le journal est une suite ordonnée d'enregistrements, dans laquelle l'écriture se fait linéairement. Dans un système résistant aux pannes qui est basé sur un journal, des informations permettant de défaire ou de refaire une transaction sont enregistrées dans le journal *avant* de les effectuer sur l'*image* des données sur disque (Principe du *Write Ahead Logging (WAL)*).

Dans cette sous-section nous ferons d'abord une liste brève des problèmes liés à la journalisation, et ensuite nous passerons en revue quelques systèmes utilisant un journal [Ruf95].

2.2.2.1 Problèmes et terminologie

La réutilisation de l'espace journal En théorie, le journal physique peut être considéré comme infini. Cependant, en réalité, le journal a seulement une taille finie, et nous devons être capable de réutiliser l'espace occupé par des enregistrements qui ne sont plus utiles. Il y a essentiellement deux solutions à ce problème :

- La compression : de temps en temps, le système décale les enregistrements vers le début de l'espace réservé au journal, en bouchant ainsi les trous laissés par les enregistrements devenus inutiles,

- Le journal circulaire : quand le système arrive à la fin de l'espace réservé au journal, il revient au début. Ceci présuppose que les enregistrements deviennent

obsolètes dans à peu près le même ordre que celui dans lequel ils ont été écrits initialement. Si ce n'est pas le cas, il faut avoir recours au *threading*, qui consiste à sauter, lors de la réutilisation de l'espace, les enregistrements qui ne sont pas encore obsolètes.

Localisation des extrémités du journal Les coordonnées des extrémités du journal (fin, et début dans le cas d'un journal circulaire) sont enregistrés dans un *enregistrement maître* qui se trouve à un endroit fixe du journal. Ceci soulève deux problèmes :

1. Si une panne survient lors de la mise à jour de cet enregistrement, on risque de se retrouver dans un état où le secteur contenant cet enregistrement est illisible : il ne contient ni l'ancienne valeur, ni la nouvelle. Ce problème est d'habitude résolu en gardant deux exemplaires de l'enregistrement maître, qui sont écrits de manière alternée (*ping-pong writes*). On a donc la garantie que, lors de la reprise, au moins un des deux exemplaires est lisible.

2. Étant donné que les enregistrements de données du journal sont écrits à des endroits variables, alors que l'enregistrement maître est écrit à un endroit fixe, les deux sortes d'enregistrements sont le plus souvent assez éloignés les uns des autres sur disque. Ceci nécessite donc des mouvements du bras du disque assez coûteux. Le système de journalisation a donc intérêt à mettre à jour l'enregistrement maître le moins souvent possible. L'enregistrement maître ne contient donc qu'une indication *approximative* de l'emplacement des extrémités du journal. Leur emplacement exact doit être déterminé lors de la reprise en suivant le journal enregistrement par enregistrement à partir de bornes approximatives. Afin de pouvoir faire ceci, les enregistrements ou les secteurs disque du journal doivent contenir des *marques* qui permettent de dire si oui ou non un tel secteur fait partie du journal ou pas. De plus, si le système utilise le *threading* pour la réutilisation du journal, il doit exister des liens de chaînage entre les différents enregistrements qui permettent de retrouver l'enregistrement suivant.

Souvent, l'enregistrement maître contient non seulement les coordonnées des bornes du journal, mais aussi les coordonnés de certains enregistrements qui ont une signification spéciale pour le protocole : des enregistrements qui résument l'état des transactions actives, qui énumèrent les données mises à jour récemment, etc.

Groupement des enregistrements Les enregistrements contenus dans le journal physique ne sont en général pas isolés, mais appartiennent à des groupes. Le plus souvent il s'agit là d'enregistrements appartenant à une transaction donnée. Mais on peut aussi imaginer d'autres critères de groupement. Par exemple, dans Camelot [DST87, Dan88] les enregistrements sont groupés par client qui les a envoyés. Ces groupes sont des entités conceptuelles, et ne sont pas nécessairement stockés de manière contiguë dans le journal. Souvent, ils sont appelés *journaux logiques*.

En plus de ces groupes logiques, les systèmes utilisent d'autres subdivisions du journal, orthogonales, qui servent à la gestion du physique du journal. Souvent ces autres subdivisions ont une localité spatiale.

Idempotence Une opération est dite idempotente si deux exécutions successives de la même opération mènent au même résultat final qu'une exécution unique. Par exemple, l'opération $a \leftarrow 3$ est idempotente, alors que $a \leftarrow a + 1$ ne l'est pas. Lors de la reprise après une panne, le système de journalisation rejoue les opérations qui n'ont *éventuellement* pas encore été faites avant la panne. Dans certains systèmes, dont le nôtre, il peut arriver que des pannes multiples mènent à des rejeux multiples de certains enregistrements. Afin d'assurer une opération correcte, tous les enregistrements doivent décrire des opérations idempotentes. D'autres systèmes, tels que Aries, rendent impossible le rejeu multiple d'enregistrements, et n'ont donc pas besoin d'imposer une telle restriction aux applications. Cependant, cette propriété est seulement acquise au prix d'un algorithme lourd et peu souple.

2.2.2.2 Camelot

Camelot subdivise son journal en *segments* de taille fixe. Un segment peut contenir des enregistrements provenant de différents clients. Chaque segment est subdivisé en quatre parties :

1. son en-tête, qui sert de «racine»,
2. les enregistrements de données,
3. un index d'enregistrements, qui répertorie la longueur et l'emplacement des enregistrements de données.
4. des descripteurs de groupe qui factorisent des informations communes à plusieurs enregistrements consécutifs (notamment le journal logique auxquels ils appartiennent)

Malheureusement, le segment est organisé de telle manière qu'il doit être écrit d'un trait, il peut donc arriver que le système de journalisation soit forcé d'écrire des segments qui sont seulement à moitié remplis quand il n'a pas assez de données à journaliser lors de la validation.

Camelot utilise une technique de *threading* par segments pour la réutilisation de son journal : si tous les enregistrements d'un segment sont obsolètes, le segment tout entier peut-être réutilisé.

Camelot permet de mettre en œuvre des journaux répartis, grâce au système DLF (Distributed Logging Facility – Service de Journalisation Réparti) décrit dans [DST87, Dan88]. Les gestionnaires de récupération sont considérés comme clients de ce service. Une des caractéristiques particulières de Camelot est son algorithme de duplication. Les enregistrements du journal sont dupliqués pour des raisons de *fiabilité* et de *disponibilité*. Supposons que nous disposions de N serveurs de journalisation en tout sur des sites différents. Les enregistrements d'un journal donné sont dupliqués sur un quorum de W serveurs parmi ces N serveurs. Les serveurs stockant un journal particulier peuvent migrer au cours du temps. Dans ce cas, les nouveaux enregistrements vont sur les nouveaux serveurs, mais les anciens restent où ils sont. Par conséquent, tous les enregistrements d'un journal ne sont pas nécessairement stockés par le même ensemble de W serveurs. Cependant, on peut obtenir

un journal complet en disposant d'un ensemble de $N - W + 1$ serveurs. Le service de journalisation de Camelot peut donc tolérer $W - 1$ pannes simultanées.

2.2.2.3 Kitlog

Kitlog [Ruf92b, Ruf92a] est un service de journalisation générique dont l'originalité est d'être personnalisable pour les besoins du client. Comme l'illustre la figure 2.4, les fonctions de stockage sont décomposées en cinq mécanismes orthogonaux, qui sont représentés par des *blocs de base* qui peuvent s'empiler et s'assembler de nombreuses façons, comme un jeu de construction :

- un *Bloc Tampon* permet de regrouper plusieurs petits enregistrements successifs en un grand enregistrement, afin d'économiser la bande passante où l'espace disque consommé ;
- un *Bloc Distribution* permet de relier plusieurs sites entre eux, afin de réaliser une architecture de journalisation à distance ;
- un *Bloc Réplication* copie un seul flot d'enregistrements en entrée sur deux flots identiques en sortie ;
- un *Bloc Partage* permet au contraire de *fusionner* plusieurs flots en un seul ;
- un *Bloc support de disque* assure la gestion du support physique, et termine ainsi la chaîne[3].

Les clients accèdent aux journaux par un bloc spécial, appelé *Vue de journalisation.*

Chacun de ces blocs est mis en œuvre par une classe d'objets. Pour chacun de ces blocs, plusieurs classes réalisant la même interface avec des mécanismes différents peuvent être fournies.

Un client assemble les différents blocs comme il le souhaite, de manière à ce qu'il soit protégé contre un ensemble particulier de pannes éventuelles qu'il a définies. Pour ce faire, le client sélectionne les classes adéquates, les instantie, et les connecte entre elles.

Tout comme Camelot, Kitlog regroupe aussi ses enregistrements dans des segments de taille fixe, qui doivent être écrits d'un coup.

À la différence de Camelot, Kitlog utilise cependant le *compactage* pour réutiliser le journal.

2.2.2.4 Clio

C'est Clio [FC87, Fin89] qui a introduit le concept de journal logique sous le nom de *fichier de journalisation.* Ce système opère sur un médium à écriture unique. Clio

3. Il existe d'autres blocs qui peuvent se substituer au bloc «support de disque», tels que le bloc support de terminal, qui affiche tous les enregistrements qui lui parviennent sur la console, et le bloc support de processus, qui communique tous les enregistrements qui lui parviennent à un processus.

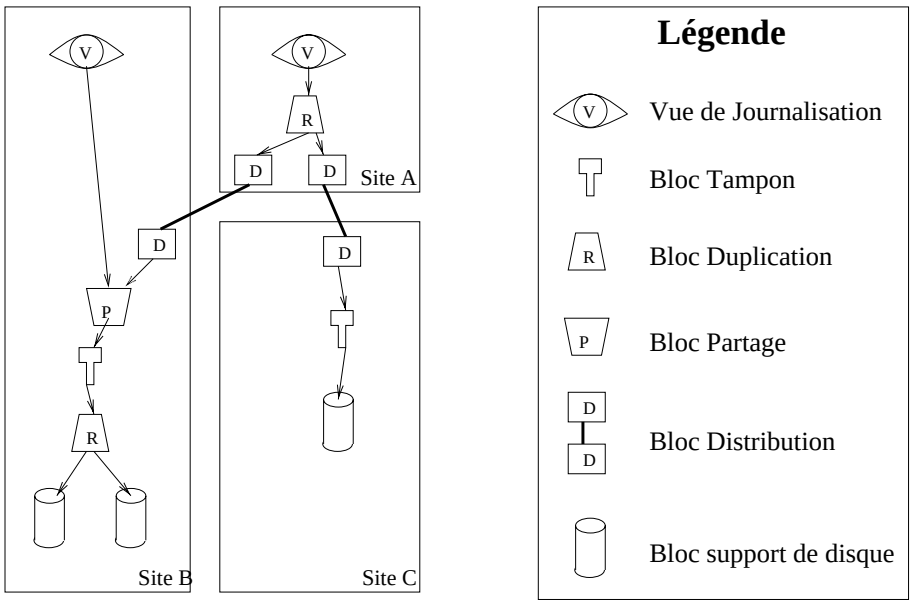

FIG. 2.4 – *Un assemblage de blocs de journalisation dans Kitlog*

offre un service de système de fichiers dans lequel les fichiers sont mis en œuvre en tant que journaux logiques.

Les journaux logiques peuvent être copiés logiquement, ou structurés de manière hiérarchique. Quand un journal est copié logiquement, aucune copie physique n'est faite, comme l'illustre la figure 2.5. Les enregistrements déjà écrits sont partagés par la source et la destination, mais à partir de la copie les deux journaux évoluent individuellement. Étant donné que dans le cadre d'un journal on ne modifie jamais des enregistrements déjà écrits, la partie partagée n'est jamais modifiée, et la sémantique d'une copie logique est donc indiscernable de celle d'une vraie copie.

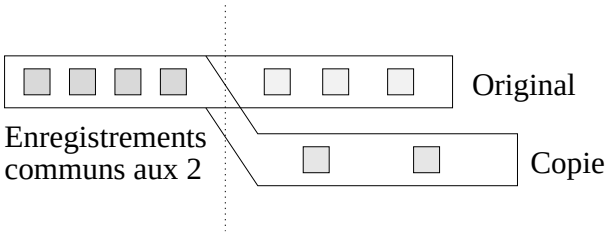

FIG. 2.5 – *Copie logique d'un journal*

Un journal logique peut aussi être créé en tant que sous-journal (partie) d'un autre. Dans ce cas, les enregistrements du sous-journal appartiennent aux 2 journaux. La figure 2.6 illustre un exemple de sous-journaux. Le journal logique «`/`» contient tous les enregistrements. Le journal logique «`/News`» contient les enregistrements correspondant aux nouvelles Usenet. Le journal logique «`/News/alt.folklore.urban`» fournit un accès au groupe de nouvelles des petites histoires folkloriques incroyables.

Chaque enregistrement est identifié par un identificateur de journal logique et une estampille qui représente l'instant où l'enregistrement a été reçu par le service

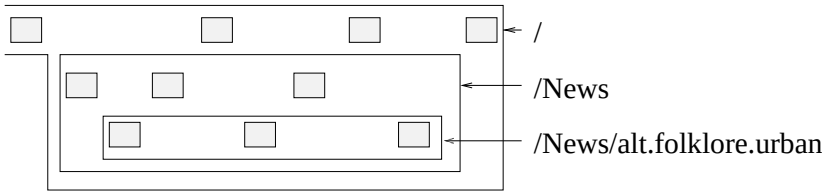

FIG. 2.6 – *Journaux hiérarchisés*

de journalisation. Pour chaque répertoire du système de fichiers de Clio, un fichier journal particulier, le *catalogue*, stocke les attributs des fichiers du répertoire (dont le nom du fichier, le propriétaire et les droits d'accès)

Comme les deux systèmes précédents, Clio subdivise son journal en segments de taille fixe. Cependant, à la différence des systèmes précédents, la structure des segments de Clio permet l'écriture progressive d'un segment : Les enregistrements (avec en-têtes) sont écrites à partir du début du segment, alors que les tailles sont écrits à la fin. L'espace qui n'est pas encore utilisé se trouve donc au milieu (figure 2.7).

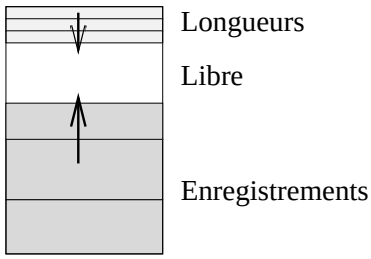

FIG. 2.7 – *Structure d'un segment en Clio*

2.2.2.5 Aries

Aries est un noyau transactionnel pour des bases de données SQL. Dans la version initiale de Aries, l'accès à la base se fait uniquement par une interface procédurale. Cependant, nous verrons dans la suite que plusieurs autres projets ont repris Aries, et lui ont ajouté une interface d'accès par mémoire virtuelle. Le grain d'accès pour la plupart des opérations dans Aries est la page. La mise en œuvre de Aries, assez complexe, sera décrite dans la suite.

Modèle de verrouillage Aries distingue deux types de verrous :

- Les verrous logiques (*locks*). Ce sont les verrous vues par l'application transactionnelle. Ils sont gardés pendant un temps assez long ; en général ils sont acquis au début de la transaction et relâchés à la fin. Ils servent à protéger les transactions les unes des autres. Ces verrous portent sur des régions.

- Les verrous physiques (*latchs*). Ces verrous sont seulement utilisés de manière interne dans Aries. Ils sont seulement gardés pendant une procédure d'accès à la base, et doivent obligatoirement être relâchés avant tout retour vers l'application. Ils portent sur une page, et servent à se protéger contre des réorganisations de données au sein d'une page ou d'un index.

Le système ne demande jamais de verrou logique quand il détient un verrou physique. Ainsi, les interblocages entre verrous physiques et logiques ne sont pas possibles. De plus, l'ordre de prise des verrous physiques garantit l'absence d'interblocage entre deux verrous physiques. La seule possibilité d'interblocage est celle qui existe entre les verrous logiques.

Organisation de l'image stable L'image stable est un ensemble de pages. Elle contient des pages de données et des pages d'index utilisées pour la localisation rapide d'enregistrements dans la base de données à partir de clefs. Chaque page comporte un champ `PageLSN`[4], qui identifie le dernier enregistrement de journal qui ait modifié cette page. Cette information sert à éviter que les modifications ne soient rejouées plusieurs fois : Parmi les enregistrements s'appliquant à cette page, seuls ceux qui sont plus récents que le `PageLSN` de cette page sont rejoués.

Organisation du journal Le journal est une suite d'enregistrements. On distingue essentiellement les deux types d'enregistrements suivants :

- Les enregistrements de données (type *update*). En général ces enregistrements sont composés de deux parties : l'une permet de rejouer la modification, et l'autre permet de la défaire.
- Les enregistrements de compensation (type *compensation* ou *compensation log record (CLR)*). Ces enregistrements visent à gérer l'annulation complète ou partielle de transactions.

Les enregistrements relatifs à une transaction donnée sont chaînés entre eux grâce aux pointeurs `PrevUndoLSN`. Ce pointeur pointe vers l'enregistrement qui est à défaire après l'enregistrement courant. En général, il s'agit de l'enregistrement précédent, sauf pour les CLR où il s'agit de l'enregistrement qui précède l'enregistrement défait par ce CLR (cf figure 2.8)

De plus, le moteur transactionnel enregistre périodiquement les informations suivantes dans le journal, dans un *point de reprise* :

- La table des pages sales, qui indique, pour chaque page modifiée, un champ `RecLSN`[5] qui est le `LSN` du premier enregistrement ayant touché à cette page[6], et qui ne soit pas encore répercuté sur l'image stable de cette page.
- La table des transactions, qui indique l'état courant de chaque transaction en cours (annulée ou validée), ainsi que le LSN de son premier enregistrement.

Un pointeur sur le point de reprise existe dans l'*enregistrement maître* du journal. Contrairement aux autres informations du journal, l'enregistrement maître est écrit à un endroit fixe (en deux exemplaires alternés). Il sert d'*ancre* de journal, et permet d'initialiser l'algorithme de reprise.

4. LSN signifie *Log Sequence Number*. C'est un numéro de séquence permettant d'identifier un enregistrement du journal.

5. Recovery LSN – LSN de reprise.

6. Ou, plus exactement le LSN du premier enregistrement écrit après le verrouillage de cette page en mémoire.

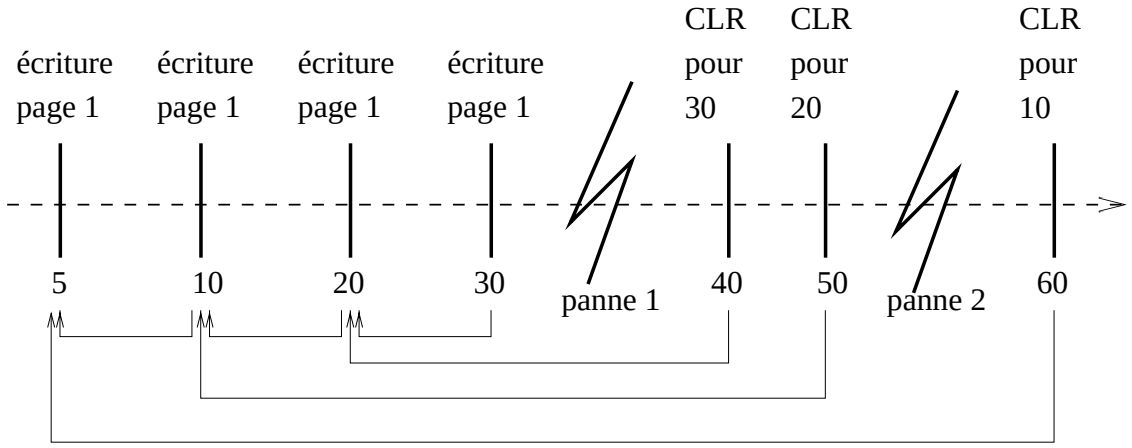

FIG. 2.8 – *Chaînage des enregistrements de compensation dans Aries*

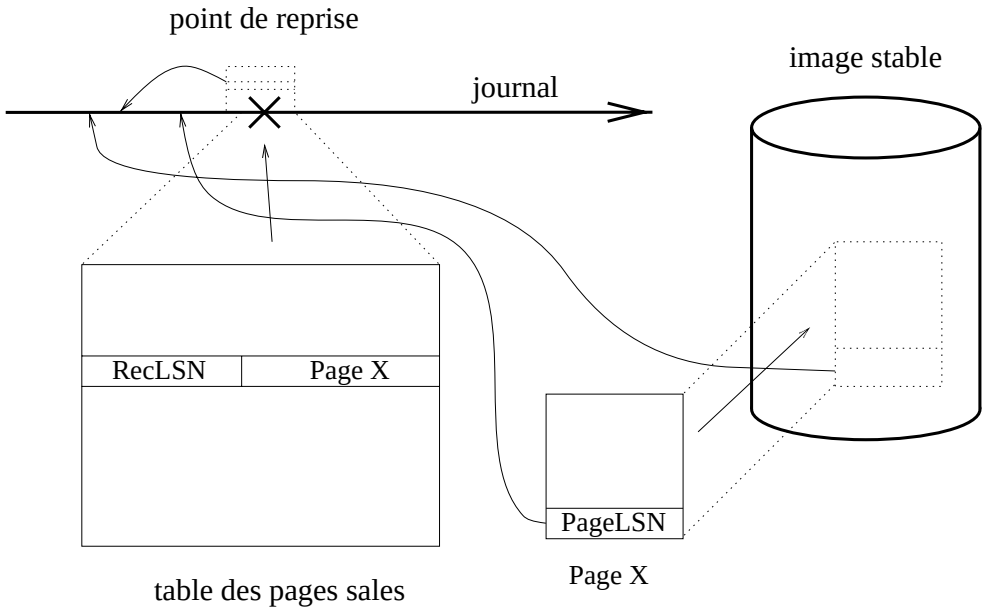

FIG. 2.9 – *Situation relative des différents LSN*

Synthèse de l'organisation La figure 2.9 illustre la relation des pointeurs entre les différentes entités dans Aries. Le `RecLSN` enregistré dans la table des pages sales du point de reprise est plus vieux que le `RecLSN` dans la copie de la page en mémoire, étant donné qu'il représente l'instant de la première modification de la page, alors que le `PageLSN` représente la dernière modification de la page jusqu'à maintenant.

Lorsque la page est écrite sur disque, elle devient propre, et son entrée dans la table des pages sales est vidée. Si de nouvelles modifications sont apportées à la page, une nouvelle entrée est créée dans la table des pages sales, ce qui donne lieu à un nouveau pointeur `RecLSN` qui sera forcément plus récent que le pointeur `PageLSN` de la copie de la page sur disque.

Le fait que, le pointeur `RecLSN` est toujours plus ancien que tout `PageLSN` (en mémoire) correspondant est utilisé pour optimiser les lectures de pages pendant la récupération.

Algorithme d'annulation intentionnelle partielle ou totale d'une transaction Une annulation «intentionnelle» d'une transaction est une annulation qui a lieu soit sur demande explicite de l'application, soit à cause d'un problème d'interblocage de verrou logique. L'annulation peut être *totale* auquel cas la transaction tout entière est défaite, ou *partielle* auquel cas la transaction est seulement défaite jusqu'à un certain instant, appelé *point de sauvegarde*.

L'annulation est faite en parcourant le journal logique de la transaction dans l'ordre chronologique inverse jusqu'au début de la transaction (annulation totale) ou jusqu'au point de sauvegarde (partielle). Pour faire ce parcours, le système utilise les pointeurs `PrevUndoLSN`, afin de sauter les enregistrements qui ont déjà été défaits lors d'éventuelles annulations partielles précédentes. Pour chaque enregistrement défait, le système rajoute un CLR au bout du journal, dont le pointeur `PrevLSN` pointe sur le prédécesseur de l'enregistrement défait. L'avantage de sauter les enregistrements déjà défaits est double :

- Cela permet d'avoir des opérations non idempotentes,
- Cela permet en outre de relâcher immédiatement les verrous logiques sur les objets verrouillés après le point de sauvegarde.

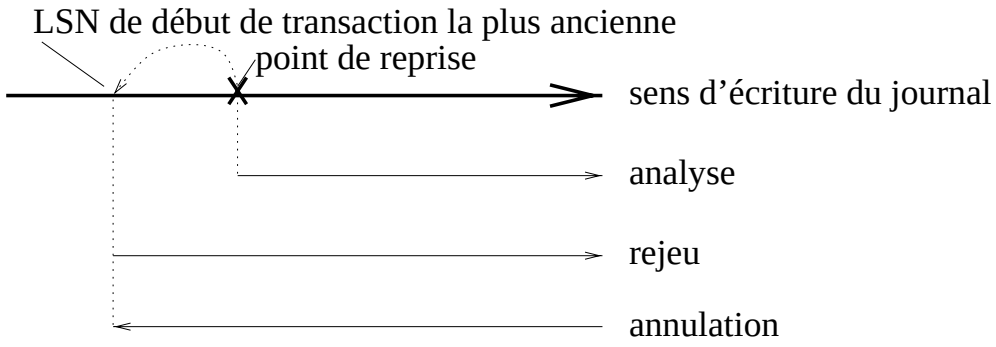

FIG. 2.10 – *Reprise après panne dans Aries*

Algorithme de reprise Aries utilise un algorithme de reprise à trois passes (figure 2.10). Dans la première passe, le système détermine l'état (validé ou abandonné) de chaque transaction qui était en cours lors de la panne. Pour ce faire, il parcours le journal dans l'ordre chronologique en partant de l'endroit consigné dans *l'enregistrement maître*. Il initialise ses tables à partir du point de sauvegarde enregistré à cet endroit, et parcourt ensuite le journal jusqu'à sa fin, tout en mettant ces informations à jour afin de tenir compte des changements d'état de transactions qui ont eu lieu après l'écriture du point de sauvegarde.

La seconde passe est une passe de reprise. Elle progresse aussi dans le sens chronologique, et elle part du début de la plus ancienne transaction en cours lors de la panne. Lors de cette phase, tout enregistrement *update* est rejoué s'il est plus récent que la page à laquelle il s'applique. Les modifications marquées dans les enregistrements *CLR* sont défaites. Les enregistrements sont rejoués (ou défaits) indépendamment de l'état de la transaction : on *refait l'histoire*. À la fin de cette passe, la base

se trouve dans exactement le même état que lors de la panne. Lors de cette phase de rejeu, les champs `PageLSN` de chaque page touchée sont mis à jour, de manière à optimiser la reprise d'une éventuelle seconde panne.

Évidemment, cet état n'est pas cohérent, en effet il contient des modifications faites par des transactions non validées. Ces modifications sont défaites par une troisième passe qui parcourt le journal à l'envers. L'algorithme d'annulation après pannes est similaire à l'algorithme d'annulation d'une transaction individuelle, à l'exception près que lors d'une annulation après panne on défait *toutes* les transactions non validées.

2.2.3 Systèmes de fichiers à base de journal (Log File Systems)

Les systèmes présentés dans cette sous-section stockent l'image des segments et le journal dans une même entité sur disque : Le système de fichiers à base de journal. L'avantage de cette approche est un gain de performances dû à une seule opération de sauvegarde plûtot que deux (journalisation et répercussion de l'image).

2.2.3.1 Sprite LFS

Sprite LFS (Log-structured File System – Système de fichiers à base de journal) [DO89, RO91] est un système de fichiers structuré comme un journal. L'idée est basée sur la constatation que, si le cache disque est suffisamment grand, la plupart des opérations disque sont des écritures. La structuration en journal permet d'optimiser la vitesse de ces écritures en les effectuant à des endroits qui sont proches les uns des autres sur le disque, et en évitant ainsi les mouvements excessifs du bras du disque.

Les fichiers sont subdivisés en pages. Quand une page est modifiée, la nouvelle version est stockée en la mettant au bout du journal physique. Ainsi, les blocs de données n'ont pas un emplacement fixe. À chaque fichier est associé un *inode* et une *table d'implantation*[7]. Les *inodes* sont stockés dans des *blocs à inodes* qui sont eux-aussi journalisés quand ils sont modifiés. Les emplacements des blocs à inodes sont enregistrés dans une table appelée la *carte des inodes*. Les différents morceaux de cette table sont aussi journalisés quand ils sont modifiés. Les emplacements des différents morceaux de cette table sont enregistrés dans un *point de reprise* qui, lui, est stocké à un emplacement fixe du disque.

Afin d'éviter de réécrire, lors de chaque modification, le chemin d'accès entier qui mène de la racine au bloc modifié (en partant du point de reprise, en passant par la carte des inodes, puis par l'inode, ensuite par les différents niveaux de la table d'implantation pour finalement arriver aux données elles-mêmes), ce chemin n'est pas mis à jour systématiquement pour chaque écriture de bloc. Les blocs récemment

7. La table d'implantation indique la position des différents blocs du fichier. D'habitude, les quelques premiers enregistrements de cette table sont enregistrés dans l'*inode*, les enregistrements suivants sont enregistrés dans des *blocs directs*, *indirects* et *doublement indirects* qui sont référencés directement ou indirectement par des pointeurs stockés eux-aussi dans l'*inode*, tout comme dans un système de fichiers ordinaire.

écrits sont plutôt localisés dans des *tables des matières (segment summary blocks)*[8] qui décrivent la nature des blocs récemment écrits. Ces tables des matières sont aussi écrites dans le journal. C'est seulement quand un bon nombre de changements des métadonnées se sont accumulés que les chemins complets sont écrits sur disque et que le point de reprise est mis à jour.

Dans Sprite LFS, les enregistrements sont des blocs disque entiers. Ils peuvent être des blocs de fichiers, des blocs de table d'implantation, des blocs d'inode, des blocs de carte d'inode ou des tables des matières.

Sprite LFS utilise des enregistrements de taille fixe et des pointeurs pour identifier ses données. Les inodes sont identifiés par un numéro d'inode unique (un index dans la carte des inodes), et les blocs sont identifiés par des pointeurs contenus dans l'inode (où dans le reste de la table d'implantation). Les identificateurs d'enregistrement, présents dans les tables des matières sont seulement nécessaires pour des raisons de performance.

L'interface d'accès à Sprite LFS est celui d'un système de fichiers Unix traditionnel.

Comme dans les autres systèmes étudiés, les enregistrements sont regroupés en segments de taille fixe. Chaque segment se termine obligatoirement par une table des matières. Cependant, des tables des matières supplémentaires peuvent figurer à l'intérieur du segment si jamais une validation survient à un instant où il n'y a pas encore suffisamment de données pour remplir un segment entier.

Pour réutiliser son espace, Sprite utilise le *threading* par segments entiers comme Camelot. En plus, il utilise le compactage s'il y a trop de segments qui sont presque vides. Le compactage se fait alors en réunissant des enregistrements de plusieurs segments creux dans un nouveau segment. Les segments à compacter sont choisis en considérant leur taux de remplissage, et aussi leur âge. En effet, les segments trop jeunes ont une probabilité plus grande de continuer à évoluer, ce qui nécessiterait un nouveau compactage des mêmes données.

2.2.3.2 Texas

Comme Sprite, Texas utilise un système de fichiers à base de journal[9]. Chaque fichier stocke l'image d'un segment. Texas résout le problème de la mise à jour du chemin d'accès de manière différente à celle de Sprite : plutôt que de différer cette mise à jour en utilisant des *segment summary blocs*, Texas met à jour le chemin le plus rapidement possible, mais s'arrange pour qu'il soit aussi court que possible. Afin de réaliser ceci, il part de l'hypothèse que la plupart des opérations sont des écritures en fin de fichier (*append*)[10]. Comme le montre la figure 2.11, les tables d'implantation sont stockées de telle manière que les chemins d'accès soient le plus court du côté droit de l'arbre. Ceci est fait en faisant dépendre les derniers blocs

8. Ces tables servent deux buts. Tout d'abord elles permettent d'identifier les blocs récemment écrits à des fins de localisation. Ensuite elles servent à déterminer, lors d'une compaction de journal, les blocs d'un segment qui sont encore utilisés.

9. Une version antérieure de Texas utilise un journal WAL classique.

10. Bien que cette hypothèse soit probablement vraie pour un système de fichiers utilisé par Unix, elle est très douteuse, à mon avis, pour une base de données.

d'un fichier directement de l'inode[11]. Ainsi, dans la plupart des cas, une mise à jour ne demande que la journalisation de deux blocs.

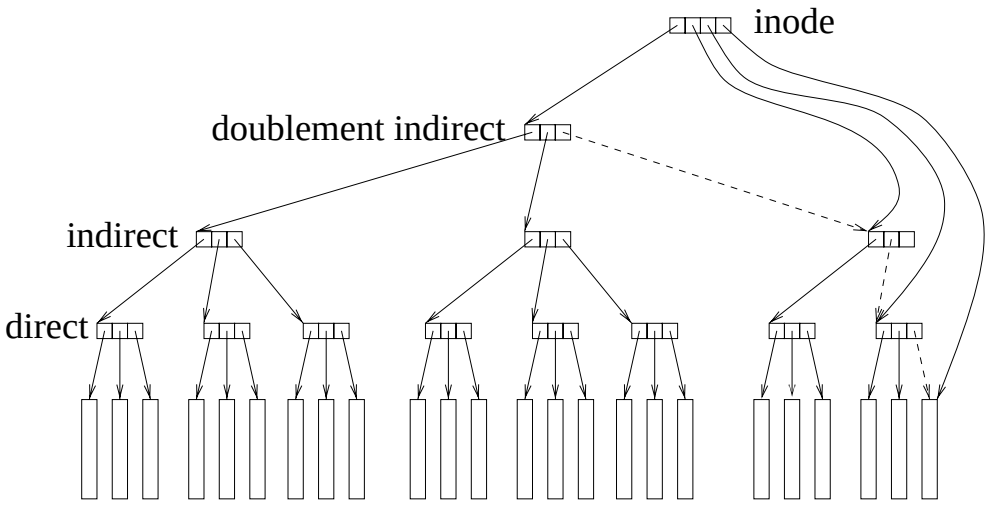

FIG. 2.11 – *Table d'implantation dans LFS*

Optimisation Afin d'optimiser le débit vers le journal, Texas utilise la technique du *Page diffing* pour mémoriser des mises à jour qui n'affectent que quelques octets par page. Lorsque l'application commence à écrire dans une page, le système garde une copie propre de la page dans un coin. Lors de l'enregistrement des modifications sur cette page, le système calcule la différence entre la copie propre et la page modifiée. S'il n'y a que quelques octets qui ont été modifiés, cette description de la différence peut être très compacte. Seule cette description de la différence a besoin d'être journalisée.

Journal physique Texas n'utilise pas de segments pour son journal physique. Il utilise une technique de threading à grain fin (secteur disque). Pour faire ceci, il utilise un champ de bits pour représenter les secteurs libres. Afin d'améliorer la vitesse d'accès il ne choisit pas le premier secteur libre, mais plutôt la première zone de secteurs où le nombre de secteurs libres est suffisamment important pour garantir un débit correct[12].

2.2.3.3 GEODE

GEODE [Ham95, BHPT93, BHPT94] est une libraire de stockage d'objets dédiée au développement d'applications transactionnelles avancées qui ne peuvent être supportées par des SGBDs traditionnels. En faisant ceci, GEODE suit le chemin

11. Cette organisation est exactement l'opposée de celle d'Unix, qui, motivée par la constatation que les accès aux premiers blocs d'un fichier sont les plus fréquents, fait dépendre les premiers blocs directement de l'inode.

12. Les disques sont plus performants quand ils lisent ou écrivent d'une traite un grand nombre de secteurs contigus que quand ils doivent ramasser les miettes un peu partout.

tracé par Exodus (voir 2.3.2), qui a soulevé l'idée des gestionnaires de stockage à usage universel.

Les points forts de GEODE se situent surtout sur le plan de sa modularité et de l'intégration avec la mémoire virtuelle, que nous aborderons plus en détail dans la section 2.3.3. Dans GEODE, la plupart des services fournis sont optionnels, et peuvent être enlevés, à condition de respecter des *contraintes de dépendance* (voir figure 2.12).

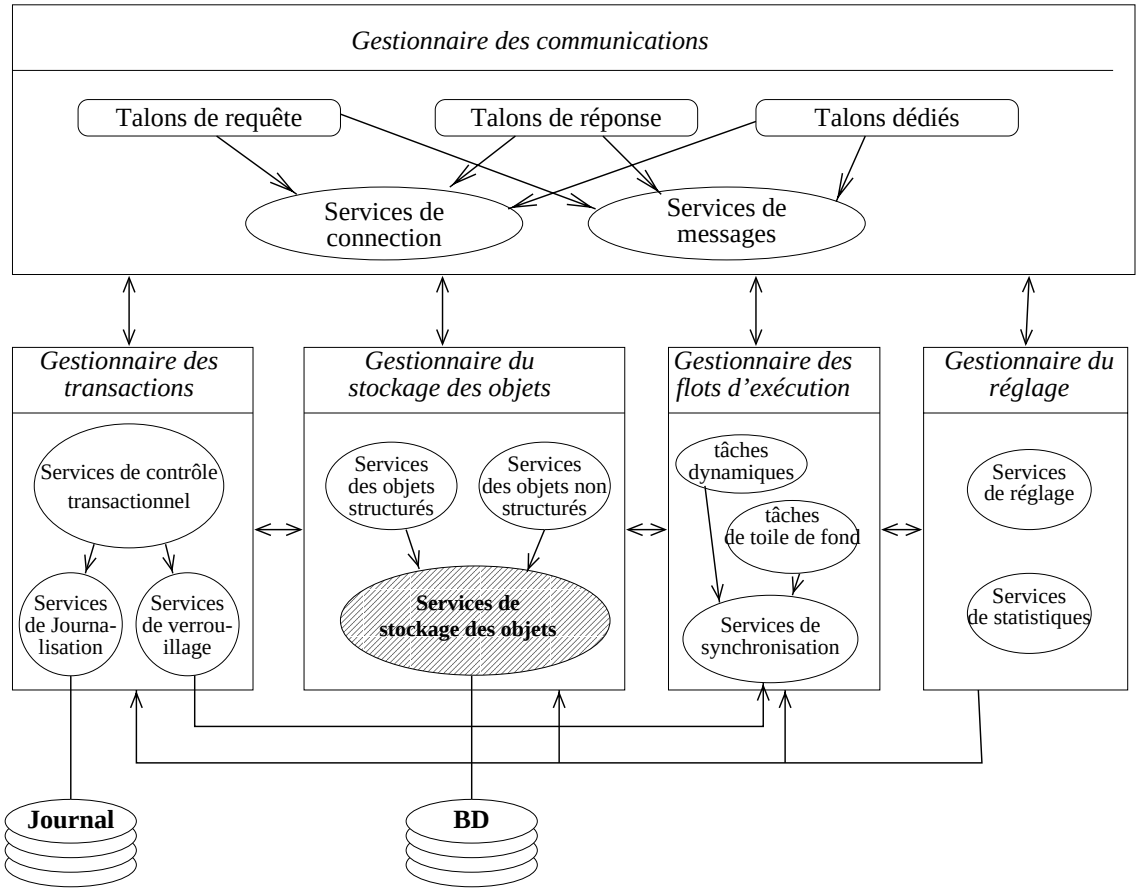

FIG. 2.12 – *Architecture fonctionnelle de GEODE*

Souplesse des transactions GEODE offre un service de stockage et de transactions qui est souple, afin de satisfaire toutes les applications visées. En effet, les applications peuvent personnaliser GEODE de plusieurs manières :

- Elles peuvent sélectivement brancher ou débrancher le service de verrouillage et/ou le service de journalisation,
- Elles peuvent manuellement introduire des images avant ou après dans le journal quand le service de journalisation automatique est débranché
- Elles peuvent gérer leur parallélisme, en utilisant un modèle de transactions basés sur des *transactions imbriquées hiérarchiques* et des *sphères de contrôle.*

Mise en œuvre du stockage Afin d'accélérer sa reprise après panne, GEODE prend périodiquement des points de reprise, en plus de ses journaux transactionnels. Pour faire ceci, il suspend toutes les applications. Pendant ce bref temps d'arrêt, il protège toutes les pages modifiées en écriture, afin de pouvoir tranquillement les sauvegarder sur disque. Cette méthode permet de limiter le temps d'arrêt au strict minimum, étant donné que la sauvegarde proprement dite peut être faite quand les exécutions ont déjà reprises. Quand une application veut à nouveau modifier une page protégée, mais non encore sauvegardée, GEODE déroule un algorithme de *copy-on-write*.

Lors de la reprise après une panne, GEODE part du dernier point de reprise complet. À partir de ce point, les transactions non validées sont défaites grâce à leur journal avant, et les transaction validées sont refaites grâce à leur journal après. Cette approche est différente de celle d'Aries, qui rejoue systématiquement toutes les transactions, y compris celle qui n'étaient pas validées (« refaire l'histoire »), avant de les défaire ensuite.

L'algorithme de reprise de GEODE est simplifié par la présence des points de reprise ponctuels (alors qu'Aries sauvegarde ses pages en continue).

2.2.4 Versions ombres (System R)

Une autre technique, pour assurer l'atomicité sont les versions ombres comme elles sont utilisées dans « System R » [GMB+81]. Cette technique est moins utilisée aujourd'hui. Elle consiste à associer à chaque segment deux versions : la version *courante* et la version *ombre*. Pendant la transaction, le système écrit les nouvelles données dans la version courante, sans écraser l'ombre. Ainsi, le système garantit qu'il existe toujours une version cohérente de chaque segment. L'opération de validation consiste juste à faire basculer les pointeurs dans la table d'implantation de manière à ce que la version courante devienne la nouvelle version ombre, et de libérer ainsi les blocs occupés par l'ancienne ombre. Cependant, cette technique offre moins de souplesse qu'un journal. Notamment, la technique de la version ombre ne permet pas, à elle seule, de faire des annulations partielles de transactions. Elle ne permet pas non plus de garantir la cohérence en présence de transactions qui modifient plusieurs segments. Afin de permettre ces opérations, « System R » utilise un journal en plus des versions ombres.

De plus, les versions ombres sont une technique coûteuse, car leur usage demande la réalisation d'une copie de chaque segment qui sera modifié. Une amélioration évidente est d'utiliser des *pages* ombres plutôt que des *versions* ombres, qui permettent de se dispenser de la copie du segment tout entier. Mais même cette variante plus économique n'est plus tellement utilisée dans les systèmes de base de données actuels.

2.2.5 Synthèse

Dans cette section, nous avons distingué plusieurs méthodes pour réaliser la permanence des données et l'atomicité des mises à jour :

– Sauvegarde et reprise (Monads, Clouds, GEODE),

- Le journal classique (Camelot, Clio[13], Kitlog, Aries[14], GEODE, System R),
- Le système de fichiers à base de journalisation (Sprite LFS et Texas),
- Les pages ombres (System R).

Ces méthodes peuvent souvent être combinées. Par exemple GEODE utilise des points de reprise périodiques et des journaux. La reprise après panne démarre alors à partir d'un point de reprise, et progresse vers l'instant de la panne en utilisant un journal.

De même, System R utilise un journal en plus de ses pages ombres, car les pages ombres à elles seules ne sont pas suffisantes pour permettre des transactions indépendantes partageant une page.

Le journal classique ou le système de fichiers à base de journalisation semblent être les deux principes les plus fréquemment utilisés. Nous avons donc consacré une importante partie de cet état de l'art à la description de ces deux techniques. Nous avons décrit la mise en œuvre d'un journal à l'aide des exemples de Camelot, Clio et Aries. L'algorithme de gestion d'Aries est de loin le plus complexe de tous les système présentés.

Nous avons parlé de Kitlog et de GEODE pour illustrer leur approche modulaire, que nous avons repris dans Arias. Kitlog permet d'emboîter ses modules les uns dans les autres, comme des tuyaux pour établir un réseau qui fait écouler les données des applications (vues) vers les supports physiques (journaux sur disque, imprimantes, terminaux, etc). Il s'agit donc surtout d'une manière pour décrire les flots de données et la répartition des données sur les différents sites, où les modules jouent le rôle de filtres et conduits de données.

La modularité de GEODE consiste à pouvoir rajouter et enlever des services entiers, qui ne sont cependant pas tous liés à la journalisation. L'agencement et le découpage des modules ne se fait donc pas de la même manière. En effet, les modules de GEODE offrent diverses fonctions du système : gestion de la synchronisation, services de contrôle de transaction, accès à des objets structurés de haut niveau, journalisation, etc.

La figure 2.13 présente une brève synthèse des points-clef des systèmes présentés.

2.3 Intégration de la mémoire virtuelle et du stockage

Cette section illustre comment l'intégration entre la mémoire virtuelle et la résistance aux pannes peut être mise en œuvre, de manière à fournir une mémoire partagée résistante aux pannes.

13. Clio utilise bien des noms hiérarchiques ressemblant à des noms de fichiers pour nommer ses journaux. Cependant, l'accès à ses journaux est purement séquentiel, donc nous rangeons ce système dans la catégorie «journal classique» plutôt que «système de fichiers à base de journalisation».

14. Les «points de reprise» de Aries contiennent juste des métadonnées, et ne constituent pas vraiment une «photo» complète et instantanée du système. C'est pourquoi nous ne rangeons pas Aries parmi les système utilisant des points de reprise.

	Monads	Clouds	Kitlog	Camelot	GEODE
Mécanisme de l'atomicité	Point de reprise	Point de reprise	Journal	Journal	Journal Point de reprise
Idée directrice			Boîte à outils (modules = filtres)	Premier système qui ait intégré mémoire virtuelle et fiabilité	Boîte à outils (modules = services)

	Clio	Sprite LFS	Texas	Aries	System R
Mécanisme de l'atomicité	Journal	LFS	LFS	Journal	Pages ombres Journal
Idée directrice	Copies logiques	LFS		Description détaillée des mécanismes d'atomicité et de reprise	Pages ombres

FIG. 2.13 – *Mise en œuvre de la journalisation et de l'atomicité*

La mémoire répartie résistante aux pannes est un concept intéressant pour deux raisons essentielles. D'abord, les programmes standard (non parallèles ou non transactionnels) peuvent bénéficier de ce système sans que l'on ait besoin de les réécrire. Ensuite, des programmes parallèles peuvent utiliser le système non seulement pour fournir la résistance aux pannes pour leur mémoire virtuelle mais aussi pour les interactions entre eux. L'aptitude à récupérer les messages en transit doit être assurée par le système de communication de messages.

La conception des systèmes à mémoire virtuelle répartie résistante aux pannes dépend de deux hypothèses. D'abord, quand une des machines participantes échoue, la panne n'est pas définitive, mais la machine peut être redémarrée, c'est-à-dire les pannes de disque ne sont pas traitées. Ensuite, nous supposons que chaque machine dispose d'un espace de stockage permanent local.

Dans cette section, nous allons illustrer, grâce à quelques exemples, comment on peut marier les concepts de mémoire virtuelle répartie et de résistance aux pannes.

2.3.1 Camelot

Camelot [Epp89] a été le premier système à fournir une mémoire virtuelle résistante aux pannes. Il est mis en œuvre au dessus de Mach. Cependant, les applications utilisateurs ne voient pas directement cette mémoire virtuelle répartie. Les applications accèdent plutôt à la base grâce à une interface procédurale classique, exportée par des modules de librairie appelés *serveurs de données*. Ces serveurs de données

sont des clients du noyau Camelot, et dialoguent avec lui en utilisant la mémoire virtuelle et des appels de procédure distant. Cette architecture est illustrée dans la figure 2.14.

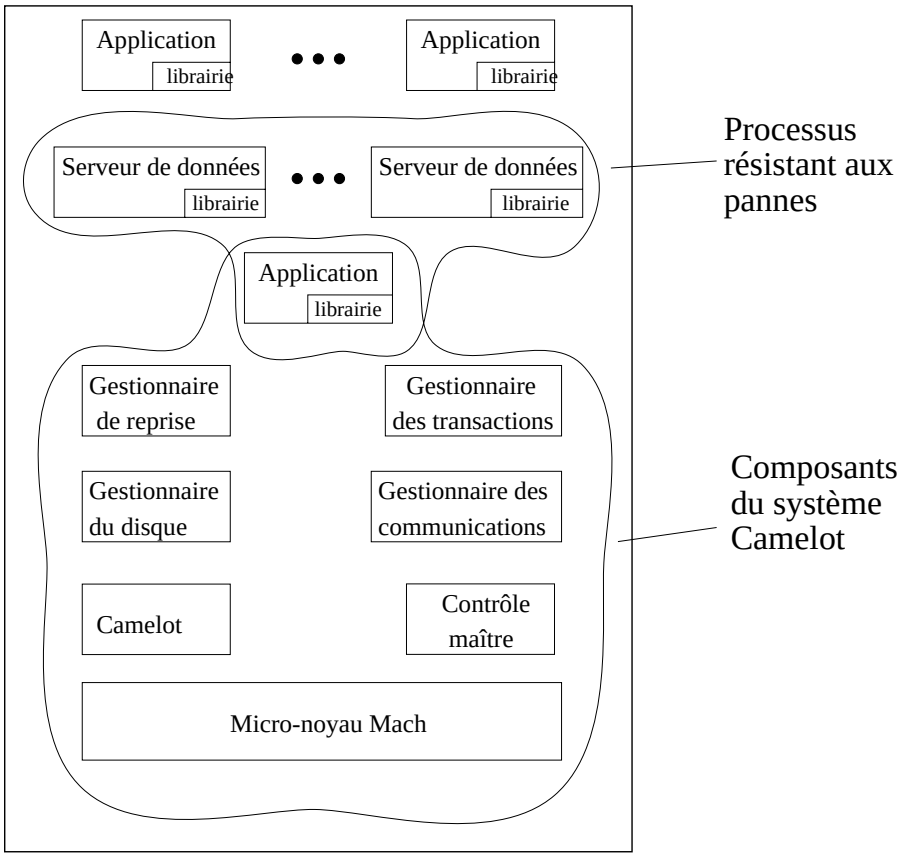

FIG. 2.14 – *Architecture de Camelot*

2.3.2 Exodus

Cette sous-section décrit le gestionnaire de stockage d'Exodus [FZT+92b], appelé ESM-CS (Exodus Storage Manager). ESM-CS offre une interface de mémoire virtuelle. ESM-CS est subdivisé en deux parties : le sous-système de journalisation, et le sous-système de récupération. Le mécanisme de reprise est basé sur celui d'Aries. Cependant, comme nous l'avons précisé dans la sous-section dédiée à Aries, Aries est plus adapté à une interface procédurale qu'à une interface mémoire virtuelle.

Exodus est un système client-serveur. Plusieurs clients couplent la base dans leur espace d'adresses, et traduisent les accès mémoire des applications en des requêtes qui sont envoyées au serveur. Les clients envoient les requêtes de journalisation (contenant suffisamment d'informations pour refaire et défaire les opérations) avant les requêtes de mise à jour de la base. Grâce à cet ordonnancement, le serveur peut plus facilement remettre en état la base quand un client tombe en panne : En effet, cet ordre assure que le serveur a toujours les informations nécessaires pour défaire les modifications sur une page donnée à partir de son état le plus récent, sans avoir besoin de rejouer le journal tout entier.

Cependant, cet ordonnancement peut avoir un effet pervers lors de l'annulation «intentionnelle» des transactions. En effet, cette annulation est effectuée par le serveur, sur les copies des pages qu'il a. Ces copies ne sont pas nécessairement à jour par rapport à celle des applications. Exodus résout ce problème en rendant *conditionnelle* l'annulation de certains enregistrements, et ne l'effectue donc que si la page présente est plus jeune que l'enregistrement à défaire. Cependant, un CLR est écrit dans tous les cas, car après la phase de rejeu déroulée après une panne, on retrouve bien entendu la page dans l'état telle qu'elle était chez le client avant la panne.

Pour savoir trouver l'endroit où commencer la phase de rejeu, et pour ne pas avoir besoin de lire systématiquement toutes les pages touchées par des enregistrements de journal, Aries utilise la *table des pages sales*, qui contient les `LSN` de l'époque à laquelle la page a été salie. Ce LSN est forcément plus vieux que les `LSN`s des enregistrements qui ont mis à jour la page depuis sa dernière sauvegarde. En commençant la phase de rejeu au minimum de tous les `LSN` de la table des pages sales enregistrée lors du dernier point de reprise, on est donc sûr (dans Aries) de ne pas rater d'enregistrement à rejouer. De manière similaire, si le `RecLSN` d'une page est plus récent que le `LSN` trouvé dans un enregistrement, ce n'est pas la peine de lire la page à partir de son image stable, car en tout cas son `PageLSN` est plus récent que son `RecLSN` et donc plus récent que le `LSN` contenu dans l'enregistrement du journal. Donc la page n'a pas besoin d'être mise à jour.

Cependant, les réalisateurs d'Exodus ont choisi de maintenir la table des pages sales chez le client, et de ne l'envoyer au serveur qu'à la fin de la transaction. La table des pages sales n'est donc pas à jour, et les optimisations réalisées dans la version initiale d'Aries deviennent donc beaucoup plus douteuses. Dans Exodus, la correction de l'algorithme est restaurée par les dispositions suivantes :

1. On introduit dans la table des transactions (sauvegardée elle aussi lors du point de reprise) le `LSN` qui était en vigueur au début de la transaction.

2. Pendant la phase d'analyse, les pages touchées par les enregistrements sont entrées dans la table des pages sales. Le `LSN` utilisé est celui du début de transaction.

En résumé, cet algorithme est assez complexe, et il aurait pu être grandement simplifié en maintenant la table des pages sales auprès du serveur : en effet ceci est possible car les enregistrements du journal sont envoyés au serveur de manière synchrone, et ils portent l'identité de la page modifiée.

2.3.3 GEODE

Le projet GEODE fut démarré au sein du projet Esprit STRETCH, et puis fut le composant central du projet Esprit IMPRESS. Il sert de plateforme à des langages orientés objet, aussi bien qu'à des langages fonctionnels. Le projet GEODE fournit des services pour stocker et manipuler des objets à différents niveaux. Dans GEODE, on distingue objets de base (qui sont vus comme des étendues de mémoire sans sémantique), des objets structurés (par exemple des n-uplets, ensembles, tableaux

et listes), ou encore des objets multimédia (images et sons) qui sont trop grands pour tenir entièrement dans la mémoire physique.

2.3.3.1 Modèle d'exécution

GEODE n'impose pas de modèle de processus et de communication. En effet, le parallélisme et la répartition peuvent s'exprimer de différentes manières au dessus de GEODE, grâce à un choix de nombreux talons client/serveur. Ainsi l'utilisateur peut choisir entre un modèle d'appel de procédures à distance et un modèle de partage d'objets.

2.3.3.2 Accès aux objets

GEODE gère son répositoire d'objets comme une mémoire virtuelle persistante, composée de segments. Les adresses des segments ne sont pas fixes, et les objets sont mobiles. Ils sont désignés par des couples (*identificateur de segment*,*identificateur d'objets*). Dans certains cas, l'identificateur de segment n'est pas nécessaire. Le système ne gère pas le typage des objets, ceci est la responsabilité du langage ou de l'application.

Cependant, malgré la présence de la mémoire virtuelle, l'accès en écriture à la base de données se fait par un modèle procédural, qui fournit des primitives de haut niveau telles que la manipulation d'ensembles. Cependant, il est aussi possible de mettre à jour un objet en fournissant une image après.

Les objets *non structurés* (de bas niveau) doivent obligatoirement être plus petits qu'une page, mais peuvent être regroupés en *objets structurés* qui dépassent la taille d'une page. De plus, le système supporte aussi de *grands objets non structurés* qui sont destinés à contenir des données multimédia. Il est possible d'accéder à ces objets en tant que *flot de données*.

2.3.4 Gestion de la répartition

GEODE utilise un modèle centralisé par segment. À chaque segment est associé un site de résidence où il est stocké sur disque et en mémoire. Si un flot d'exécution d'un autre site veut y accéder il a deux options :

- L'approche *function shipping* : le flot d'exécution fait un appel de procédure à distance (ou plutôt de méthode à distance) vers le site de résidence du segment,

- L'approche *data shipping* : un objet proxy est créé sur chaque site ou s'exécutent des utilisateurs de ce segment. La bibliothèque GEODE déroule alors entre ces proxies et la copie maître un protocole de mémoire virtuelle répartie dont le grain est l'objet. Cette option est plus efficace qu'un algorithme de mémoire virtuelle répartie s'appuyant sur des fautes de pages matérielles car elle permet de bénéficier d'informations fournies par le service de verrouillage.

2.3.5 Shore

Un de ses buts de Shore [CDF+94] est de présenter la base de données de deux manières : d'une part comme une base de données orientée objet traditionnelle, et d'autre part comme un système de fichiers Unix. En quelque sorte, la base contient deux sortes d'objets : les objets *enregistrés* qui portent un nom dans l'arborescence Unix, et les objets *anonymes*, qui sont seulement accessibles en suivant des pointeurs à partir des objets enregistrés. Les objets peuvent avoir un champ *texte* accessible par les appels système `read` et `write` de Unix. L'intérêt de cette présentation est de simplifier le portage des applications prévues pour opérer sur des fichiers texte. Ainsi on peut prévoir une transition continue d'un système basé sur des fichiers vers un système basé sur une base de données orientée objet, en adaptant à chaque pas qu'une petite partie du corps applicatif.

Le but de Shore est d'être indépendant du langage utilisé (par exemple C++ et Clos). Cependant, tous les exemples donnés dans le papier sont présentés en C++. La gestion des objets sales n'est pas mise en œuvre en utilisant le mécanisme de mémoire virtuelle, mais plutôt par des méthodes spéciales attachées aux objets. Par exemple, en C++, cette méthode s'appelle *update*, et doit être invoquée par les autres méthodes si celles-ci veulent mettre à jour des champs de l'objet. Par défaut, tous les pointeurs sur des objets portent l'attribut `const`, et la fonction *update* retourne un pointeur sans cet attribut. Ceci permet au compilateur de vérifier si le protocole de mise à jour est bien respecté, car le compilateur interdit de modifier le contenu d'un pointeur `const`.

Shore est aussi un système réparti : il peut y avoir plusieurs machines clientes, mais aussi plusieurs machines serveurs. Le système de stockage sous-jacent est Aries. Shore doit donc faire face aux mêmes problèmes que Exodus, mais les résout de manière beaucoup plus élégante : toutes les mises à jour de pages sont refaites chez le serveur, à partir des enregistrements de journal qu'envoie le client. Il n'y a donc jamais de transfert de pages sales du client vers le serveur, car la mise à jour de ces pages se fait en parallèle chez le serveur et le client. Les pages tampon du serveur, et sa table des pages sales sont donc toujours à jour, ce qui évite à Shore de faire les mêmes contorsions qu'Exodus.

2.3.6 Cricket

Les applications visées par le système Cricket [SZ90] sont les bases de données CAO (conception assistée par ordinateur). Ces bases de données donnent lieu à des caractéristiques d'accès différentes des SGBD traditionnels. Les solutions traditionnelles sont donc inacceptables pour ce genre d'application. Il s'agit essentiellement d'un problème de performance : une application de CAO tournant sur un système de SGBD traditionnel a besoin d'écrire un enregistrement de journal pour chaque modification élémentaire, ce qui a un impact négatif visible sur le temps de réponse.

Modèle de mémoire et mise en œuvre Cricket utilise un modèle à mémoire uniforme. Ce système est mis en œuvre au-dessus de Mach, et utilise les *objets mémoire (memory objects)* [Y+87] pour coupler la base de données en mémoire. Comme montré dans la figure 2.15, Cricket utilise un modèle client-serveur.

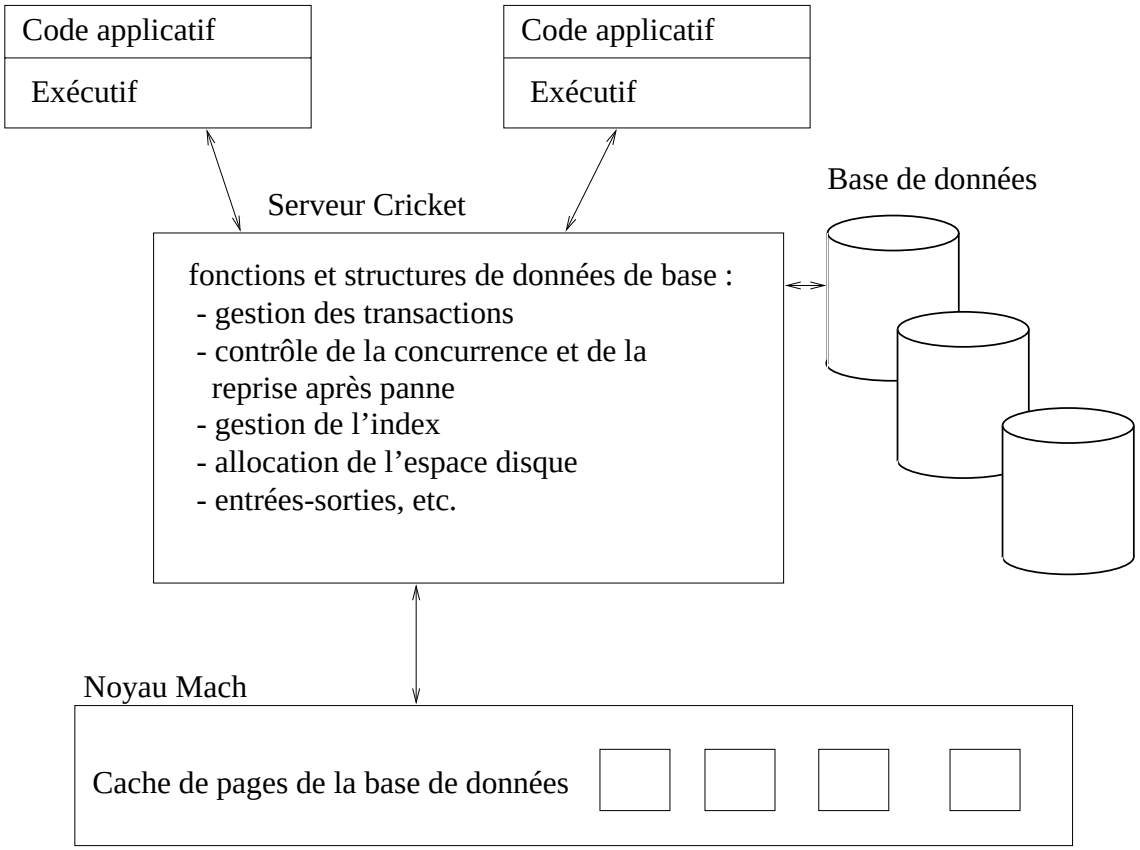

FIG. 2.15 – *Architecture de Cricket sur un site*

Gestion de l'exclusion mutuelle Pour gérer l'exclusion mutuelle entre des transactions concurrentes, Cricket utilise le mécanisme de traitement d'exception de Mach qui permet de faire traiter une exception par un autre processus que celui qui l'a déclenchée. Quand une application commence une transaction, Cricket verrouille, pour cette application, toute la plage d'adresses correspondant à la base de données. Tout accès à cette plage déclenche une exception, qui est interprétée par le serveur Cricket comme une demande de verrou. Ceci implique que le grain de verrouillage dans Cricket est la page. Cette méthode de verrouillage a été initialement proposée par le système de base de données Bubba [BAC+90].

Stockage de l'image Les données sur disque sont regroupées en des *étendues* de taille fixe. Ces étendues sont essentiellement utilisées pour la gestion de l'espace physique sur disque, et elles n'ont aucune signification de haut niveau. Pareillement, la mémoire virtuelle est subdivisée en étendues de même taille. Pour traduire une adresse virtuelle en une adresse disque, elle est d'abord décomposée en une partie étendue et une partie déplacement. Une table de hachage permet de faire correspondre une étendue sur disque à l'étendue mémoire. Le déplacement est le même sur disque et en mémoire.

Afin d'améliorer la localité des références, une application peut demander, lors de l'allocation d'un nouvel objet, qu'il soit alloué dans la même page (ou sinon dans la même étendue) qu'un objet existant.

2.3.7 Synthèse

Dans cette section nous avons décrit des systèmes qui offrent un modèle de mémoire uniforme aux applications tout en disposant de transactions fiables.

Les systèmes se distinguent d'abord par l'interface offerte aux applications. Une interface procédurale rend la mise en œuvre de la journalisation plus facile, car il suffit d'insérer les talons de journalisation dans les procédures de mise à jour. Une interface à base de mémoire virtuelle facilite la mise en œuvre des applications. Une interface à base de systèmes de fichiers facilite l'adaptation progressive des applications existantes. Conformément au sujet de cette section, tous les systèmes présentés offrent entre autres une interface à base de mémoire virtuelle. Mais celle-ci peut être accompagnée d'autres mécanismes :

- Dans Camelot, la mémoire virtuelle n'est utilisée que par les serveurs intermédiaires (librairies «serveurs de données»), alors que le code applicatif utilise une interface procédurale.

- GEODE fournit les deux : l'interface à base de mémoire virtuelle est utilisée pour les accès en lecture, alors que l'interface procédurale est utilisée pour les mises à jour. Cependant, une primitive de «remplissage» d'objets est disponible, qui permet de modifier un objet en fournissant une image après.

- Shore fournit une interface «système de fichiers» en plus de son interface directe. Cette interface sert surtout à faciliter l'adaptation d'applications écrites pour les systèmes traditionnels.

Certains des systèmes présentés, comme par exemple Exodus, Shore et dans une certaine mesure GEODE s'appuient sur l'algorithme de journalisation de Aries. Cependant, à cause des contraintes particulières (répartition, interface mémoire virtuelle), le modèle de Aries a dû être adapté. La répartition a posé des problèmes de cohérence entre le journal et l'image permanente. Exodus et Shore proposent deux solutions différentes à ce problème, et la solution de Shore me paraît nettement supérieure. L'algorithme de GEODE est suffisamment différent de celui de Aries (absence du concept de «refaire l'histoire») pour que le problème ne s'y pose pas.

Deux des systèmes présentés (Camelot et Cricket) sont construits au-dessus de Mach. En effet, ce micro-noyau a des primitives de gestion de mémoire virtuelle (*objets mémoire*) faciles à utiliser, ainsi qu'une gestion d'exceptions souple : les exceptions redirigeables de Mach sont utilisées par Cricket pour piloter le système de verrouillage. Malheureusement on paye ces avantages par des performances assez faibles, de façon que Mach est de moins en moins utilisé.

La figure 2.16 présente une brève synthèse de certains points-clef des systèmes présentés.

2.4 Conclusion

Dans cet état de l'art nous avons présenté un bref aperçu des systèmes à mémoire virtuelle, ainsi que des techniques de journalisation. Plusieurs systèmes, tels

	Camelot	Exodus	Geode	Shore	Cricket
Interface	Procédurale et mémoire virtuelle	Mémoire virtuelle	Procédurale et mémoire virtuelle	Mémoire virtuelle et système de fichiers	Mémoire virtuelle
Stockage	?	Aries	Variante de Aries qui ne rejoue pas les transactions abandonnées	Aries	Système à base d'"étendues"
Synchronisation et verrouillage	?	?	?	Langage	Synchronisation implicite déclenchée par les accès mémoire

FIG. 2.16 – *Intégration stockage et mémoire virtuelle*

que *Monads, Clouds, Exodus* ont tenté de réunir ces deux aspects. Cependant, la plupart des systèmes que nous avons passés en revue offrent une politique de cohérence unique pour leur mémoire virtuelle, et/ou une politique de stockage unique. Nous nous proposons dans Arias de réaliser une mémoire virtuelle répartie *souple* et *extensible* qui autorise les applications à personnaliser le système à leur goût.

Nous retenons les aspects suivants des systèmes discutés :

- Nous offrons une mémoire virtuelle à 64 bits d'adresse, tout comme Opal, Mungi et Angel ;
- Comme Aries, nous utilisons un journal . Cependant, la mise en œuvre du journal de Aries nous a paru trop compliquée, et nous avons opté pour une solution plus simple et plus souple ;
- Notre système est modulaire, tout comme Kitlog.
- Notre découpage en modules ressemble davantage à celui de GEODE qu'à celui de Kitlog. Mais contrairement à GEODE, nous descendons notre système dans le noyau, plutôt que d'utiliser une librairie en mode utilisateur. Ceci nous permet d'interagir avec la mémoire virtuelle et d'utiliser un mécanisme qui s'appuie à la fois sur des fautes de page matérielles et sur des informations fournies par le service de verrouillage. En nous appuyant sur la mémoire virtuelle, nous pouvons optimiser ainsi les opérations de sauvegarde. Cependant, le fait de placer notre système dans le noyau à rendu la réalisation et la mise au point un peu plus fastidieuse.

3

Présentation générale de Arias

3.1 Objectifs

L'objectif du projet Sirac est de concevoir et de réaliser des outils et services pour les systèmes qui nécessitent le partage d'informations complexes, structurées et à longue durée de vie. Les applications visées par ces systèmes se caractérisent plus particulièrement par :

- Une grande taille : la mondialisation des bases d'information implique des applications gérant des volumes très importants de données,
- La persistance des données gérées : les données produites par une application survivent à la terminaison de celle-ci et peuvent être réutilisées par d'autres applications,
- Des relations matérialisées par des liaisons multiples entre les données. Ces grands volumes de données doivent nécessairement, pour être utilisables, être structurés en éléments se désignant entre eux,
- Un fort taux de partage entre des utilisateurs coopérants,
- Un accès depuis des stations de travail réparties.

Des applications typiques concernent la manipulation de documents hypertextes ou hypermédias de grande taille dans les domaines tels que la CAO ou le génie logiciel. Notons également que les SGBD ont des objectifs qui correspondent bien au cadre énoncé ci-dessus.

Nous avons également pu observer un consensus autour du système Unix qui constitue aujourd'hui un standard de facto, bien que ce système se révèle inadapté au support de ces applications par bien des aspects :

- Mécanismes rudimentaires pour le partage d'information,
- Pas de gestion intégrée de la répartition, en particulier pour le nommage,
- Gestion de la persistance laissée à la charge de l'utilisateur.

Enfin, les évolutions techniques avec notamment l'arrivée des processeurs à adressage 64 bits et des réseaux ATM, permettent une remise en cause des solutions proposées pour la mise en œuvre des applications. Les réseaux rapides rendent plus réalistes les solutions fondées sur la gestion d'une mémoire virtuelle répartie et les grands espaces d'adressage promeuvent la gestion d'un espace virtuel unique.

Nous nous proposons donc de fournir un ensemble de services systèmes qui se caractérisent comme suit :

- Ces services doivent composer un support adapté pour le partage d'informations complexes et persistantes. Ils doivent à la fois répondre aux besoins des applications coopératives et des gestionnaires de base de données.

- Ces services ne doivent pas rompre avec les standards du domaine, et principalement le système Unix.

- Ils doivent tirer parti des récentes évolutions techniques pour être efficaces.

Les services à réaliser visent un large spectre d'environnements d'exécution existants pouvant en tirer parti (des exemples sont donnés en section 3.7). Bien qu'il soit concevable de réaliser directement des applications sur ces services, ce n'est pas notre objectif principal. Nous visons plutôt à fournir des services adaptés au support d'environnements, un environnement étant dédié à une classe d'applications ayant des comportements similaires. Des exemples déjà cités sont les environnements de développement d'applications réparties ou les SGBD. Si de nombreux aspects du système habituellement cachés sont rendus visibles par l'interface des services fournis, ces aspects seront certainement masqués par la plupart des environnements supportés. Dans la suite, nous utilisons le terme « application » pour les entités clientes (environnement ou application finale).

Après une présentation générale des services fournis (section 3.2), nous présentons respectivement nos propositions pour la gestion de la mémoire (section 3.3), la cohérence et la synchronisation (section 3.4) et la permanence (section 3.5).

La section 3.6 propose une architecture de réalisation, la section 3.7 les validations prévues et la section 3.8 une comparaison avec les projets similaires.

3.2 Présentation générale de l'architecture

Nous voulons fournir un service permettant de partager des données persistantes entre des processus Unix qui s'exécutent sur des stations de travail homogènes interconnectées (cf figure 3.6). Trois types de problèmes se posent :

- Comment réaliser le partage et garantir la cohérence des données partagées ? Si le partage est mis en œuvre par copie des données partagées sur les différentes machines, il se pose le problème du type de cohérence à garantir entre les copies.

- Comment résister aux pannes ? Les données partagées sont persistantes, c'est à dire que leur durée de vie ne dépend pas de celle du processus qui les a

créées. Cependant, cette propriété ne suffit pas à garantir la permanence de ces données en cas de panne. Il faut donc fournir un service assurant la permanence et la remise à disposition des données après une panne d'une des machines.

– Comment protéger l'accès aux données partagées ? Il faut permettre des accès aux données partagées avec des droits pouvant être différents d'un processus à un autre et donc fournir un service de protection d'accès aux données partagées.

Les services que nous proposons prennent donc en compte ces trois types de problèmes. Cependant, nous choisissons de donner aux utilisateurs la liberté de rendre permanentes les données persistantes partagées qu'ils utilisent et de protéger l'accès à ces données. Cela impose donc de fournir et de mettre en œuvre les services de permanence et de protection de manière indépendante.

Un second choix est de ne pas imposer de politique aux utilisateurs des services mais plutôt de fournir des mécanismes permettant à l'utilisateur de mettre en œuvre la politique qui correspond le mieux aux besoins de son application. Par exemple, le système n'impose pas de protocole de cohérence mais fournit les outils et les mécanismes permettant de mettre en œuvre différents modèles de cohérence des données persistantes partagées. Les concepteurs d'applications pourront également choisir d'utiliser les politiques usuelles qui sont fournies.

Nous donnons dans la suite une description générale du service de partage de données proprement dit, du service assurant la permanence et enfin du service de protection.

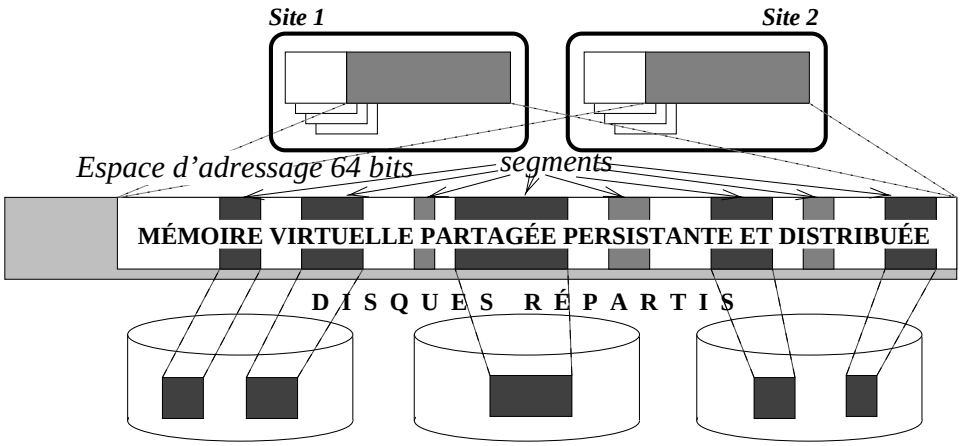

FIG. 3.1 – *Support mémoire*

3.2.1 Service de partage des données

Le service de partage de données comprend deux parties : une partie gestion de la mémoire partagée (allocation, destruction) et une partie gestion de la cohérence.

3.2.1.1 Gestion de la mémoire partagée

La mémoire partagée gérée par notre système est constituée d'un bloc de 2^{63} octets réservé à la même adresse virtuelle dans la mémoire de tous les processus qui

l'utilise. Notre service gère donc un espace virtuel unique.

L'unité d'allocation de cette mémoire est le segment qui est une suite de pages contiguës. La taille d'un segment est fixée à sa création. Les segments sont gérés de manière persistante (leur durée de vie n'est pas liée à celle du processus qui les a créés). Un segment est désigné par son adresse qui reste inchangée pendant toute sa durée de vie. Les segments doivent être explicitement détruits. Le système garantit que la portion de mémoire occupée par un segment détruit ne sera jamais réallouée et que toute tentative d'accès à un segment détruit donnera lieu à une erreur. La mise en œuvre d'un « ramasse-miettes » est laissée à la charge des applications qui seules peuvent interpréter le contenu des segments.

A l'exécution, la gestion de la mémoire partagée repose sur les mécanismes de pagination des machines. En raison du partage des segments, des accès concurrents à une même page peuvent être effectués sur plusieurs sites simultanément, cette page étant dupliquée dans la mémoire de chacun des sites. Le système ne gère pas la cohérence entre les différentes copies d'une même page mais il fournit des mécanismes permettant aux applications de mettre en œuvre leur propre politique.

3.2.1.2 Gestion de la cohérence

Les mécanismes offerts reposent sur les constatations suivantes :

- Le segment et même la page sont des unités de grain trop gros pour être choisies comme unité de contrôle d'accès (en terme de synchronisation) et de cohérence. Il est donc nécessaire de fournir une unité de grain plus fin si on veut éviter les problèmes dus au faux partage (processus accédant à des zones disjointes de la même page).

- Les applications partageant des données potentiellement accessibles simultanément par plusieurs processus synchronisent leurs accès à ces données. Il est donc possible de lier la synchronisation et la mise en cohérence des données et notamment de retarder l'application du protocole de mise en cohérence d'une zone de données lors de la demande de verrou sur cette zone.

Les mécanismes de mise en cohérence gèrent des zones qui sont des suites d'octets contigus. À tout instant le système sait localiser la copie maîtresse de la zone et fournit des primitives permettant de mettre à jour une copie de zone à partir de sa copie maîtresse, ou de changer le site de résidence de la copie maîtresse d'une zone. Ce jeux de primitives permet de mettre en œuvre les politiques classiques de gestion de cohérence (cf section 3.4). Par ailleurs, nous fournissons un certain nombre de protocoles normalisés, comme par exemple le protocole de cohérence faible (*entry consistency*), où la zone est mise en cohérence forte lors de la prise de verrou d'accès. Cette gestion de cohérence peut soit être accompagné d'une politique de synchronisation de type lecteurs/rédacteurs, soit d'une politique de verrouillage à deux phases transactionnelle.

3.2.2 Service de permanence

Un segment de cette mémoire peut être rendu permanent (avoir une image sur un support permanent, disque ou autre) ce qui lui permet de résister aux pannes du système. A l'opposé, les segments non permanents sont dits volatils. Deux mécanismes sont fournis pour la gestion de la permanence. Le premier permet de rendre permanente l'image d'un segment (le segment passe alors de l'état volatil à l'état permanent). L'atomicité est offerte par le deuxième mécanisme, en l'occurrence la journalisation. Le système permet de stocker des informations dans des journaux gérés sur disque. L'utilisateur peut enregistrer une liste de modifications dans un journal, puis après validation du journal répercuter ces modifications sur l'image permanente de la mémoire. En cas de panne, le journal permet de reconstruire une image cohérente des modifications qui y ont été validées. Les modifications non validées dans le journal sont perdues. Remarquons que le format des enregistrements de modification et les opérations de répercussion de ces modifications sont effectuées sous le contrôle des applications. En effet, les applications contrôlent le format et la sémantique des enregistrements écrits dans le journal, ainsi que l'opération d'exploitation du journal lors de la reprise d'un site après une panne. Une application peut ainsi gérer un journal «avant» (sauvegardant les anciennes valeurs validées) ou «après» (mémorisant les modifications validées). L'application contrôle également la mise à jour des images permanentes des segments qui peut être effectuée de manière asynchrone implicitement ou synchrone explicitement (cf. section 3.5).

Dans notre système, nous distinguons donc deux niveaux de mémoire : la mémoire d'exécution (dans laquelle les segments sont manipulés par les applications) et la mémoire permanente. Cette distinction permet de toujours maintenir, même pendant l'exécution, une version cohérente des segments en mémoire permanente.

Pour faciliter leur administration, les images permanentes des segments sont regroupées dans des unités logiques de stockage appelées volumes. Un volume est géré par un seul serveur à un instant donné. Il peut néanmoins être déplacé vers un autre serveur au cours de son existence. Pour assurer une plus grande disponibilité des segments, les volumes gérés par un serveur pourront être dupliqués sur d'autres machines.

3.2.3 Service de protection

Les concepts sur lesquels reposent le service de protection sont ceux de domaine de protection et de capacité. Un domaine de protection définit un ensemble de segments accessibles et pour chacun de ces segments les opérations permises. Par ailleurs un mécanisme assure que les processus s'exécutant dans le domaine accèdent à ces segments en respectant les droits définis. Une capacité est une structure de données qui intègre le nom d'un segment (c'est à dire dans notre cas une adresse virtuelle) et un ensemble de droits d'accès à ce segment. Les droits sur un segment sont exprimés en termes des opérations lire, écrire et exécuter.

La mémoire virtuelle distribuée est partagée par l'ensemble des processus Unix utilisant le service de données partagées. À un instant donné, un processus Unix utilisant le service de protection s'exécute dans un domaine de protection qui définit son contexte courant d'adressage de la mémoire virtuelle partagée répartie. Deux

domaines de protection différents peuvent partager des segments avec les mêmes droits ou des droits différents. Lors de la première tentative d'accès à un segment dans un domaine, le système vérifie si le domaine possède une capacité sur le segment et, si c'est le cas, autorise le couplage du segment avec les droits spécifiés par la capacité.

Un domaine peut exporter vers les autres domaines des capacités d'un type spécial, appelées capacités de changement de domaine, qui permettent d'appeler un sous ensemble des procédures qu'il contient. Les procédures exportées sont appelées points d'entrée d'un domaine. La tentative d'appel dans un domaine D d'un des points d'entrée du domaine D' provoque le changement de domaine du processus qui a effectué l'appel si D contient une capacité de changement de domaine autorisant l'opération. Un changement de domaine s'accompagne en général d'un transfert de capacité du domaine appelant vers le domaine appelé pour les paramètres d'appel et du domaine appelé vers le domaine appelant pour les résultats. Le service de protection fournit un langage d'interface (IDL) qui permet de spécifier la nature des capacités transférées de l'appelant vers l'appelé et réciproquement. Les transferts de capacités sont spécifiés par le domaine qui exporte ses points d'entrée et doivent être acceptés par les domaines appelant. L'avantage essentiel de notre approche est de permettre une définition du code complètement indépendante des domaines de protection dans lesquels il peut s'exécuter.

Pour permettre la définition et la structuration des domaines de protection, le système fournit la notion de liste de capacités. Les listes sont partageables entre les domaines et protégées par des capacités.

Dans la suite, nous étudions essentiellement les quatre axes de base nécessaires à la réalisation du service mémoire que nous concevons. Il s'agit de la gestion des segments, de la cohérence et de la synchronisation, de la permanence, et de la protection.

3.3 La gestion des segments

Nous avons vu précédemment que la mémoire persistante gérée par le système est composée de segments. Un segment est implanté à la même adresse virtuelle durant toute sa vie et garde aussi la même taille, celle de sa création.

3.3.1 Manipulation des segments

3.3.1.1 Création

Pour créer un segment, une application indique la taille du segment à allouer, et reçoit en retour l'adresse de début du segment créé. Cette création a pour conséquence la réservation dans l'espace virtuel d'une plage d'adresses d'une taille suffisante. Le segment garde par la suite sa taille de création ; il ne peut pas grandir.

Au moment de la création, seul l'espace virtuel est alloué. Les pages de données qui composent le segment ne sont créées qu'au moment où l'application y accède.

3.3.1.2 Destruction

La destruction d'un segment peut être explicite si une application appelle la fonction de destruction, ou implicite, lorsque toutes les capacités sur ce segment ont disparu du système. Dans cette dernière situation, en effet, aucun domaine n'a le droit d'accéder au segment : il est donc détruit. La destruction explicite est immédiate et irrévocable, mais elle nécessite un droit particulier.

La destruction d'un segment implique celle de toutes les structures afférentes, y compris l'image permanente, si le segment avait été rendu permanent auparavant. Après la destruction du segment, la plage d'adresses virtuelles qu'il occupait devient invalide. Or, certaines applications peuvent posséder des pointeurs vers cette plage d'adresses, ou certains segments peuvent contenir de tels pointeurs. Un accès au segment détruit provoque une exception chez l'application demandeuse.

3.3.1.3 Accès

L'accès par une application à une page qui n'a jamais encore été utilisée provoque un défaut de page. Pour traiter ce défaut, le système commence par identifier le segment auquel la page manquante appartient. S'il s'avère que cette page n'appartient à aucun segment (le segment n'a jamais été alloué ou a déjà été détruit), un signal d'exception est transmis à l'application. Dans le cas contraire, le segment, s'il ne l'est pas déjà, est couplé sur ce site. Ensuite, une copie de la page est amenée en suivant les règles de cohérence (cf. section 3.4).

Notons que si l'application utilise un protocole de cohérence et de synchronisation particulier, ce dernier s'occupera de l'installation de la page avant que l'application ait eu le temps de provoquer un défaut de page. En fait, ce cas constitue le comportement normal, puisque les applications sont sensées utiliser un protocole de synchronisation pour accéder à des données partagées.

3.3.2 Localisation

Nous nous posons ici le problème de la localisation des segments dans notre mémoire virtuelle. Rappelons que nous distinguons deux niveaux de mémoire : la mémoire d'exécution et la mémoire permanente. Un segment est en mémoire d'exécution s'il est utilisé par au moins un processus. Si un segment permanent n'est pas en mémoire d'exécution, il faut le localiser en mémoire permanente, puis créer les structures de données qui le représentent en mémoire d'exécution. Notons qu'un segment volatil n'existe qu'en mémoire d'exécution.

Un segment est représenté en mémoire d'exécution par un descripteur de segment. Ce descripteur contient les informations nécessaires à la gestion de la cohérence des données. Ces informations sont utilisées pour répondre à une faute de page. L'hypothèse de travail initiale est que ce descripteur est géré en version unique sur un seul site, le **site primaire**, et que tous les autres sites qui utilisent aussi le même segment doivent communiquer avec le site primaire pour accéder au descripteur. Plus loin, nous dépasserons cette hypothèse en introduisant des caches.

Lors d'un défaut, il faut donc retrouver le site primaire. Une fois cela fait, le descripteur est utilisé par le protocole de gestion de cohérence.

Pour la localisation des descripteurs (que l'on appelle localisation des segments), nous mettons en œuvre les mécanismes suivants :

- Une table de partition. L'espace virtuel global est découpé en **partitions** de même taille. Une partition est ensuite assignée à son **site de localisation**. La table de partition permet de trouver le site de localisation d'un segment à partir de son adresse virtuelle. Notons qu'un site peut localiser plusieurs partitions, et que les partitions peuvent être créées au fur et à mesure que l'espace virtuel global est utilisé.

- Pour chaque site qui localise au moins une partition, il existe une **table des segments localisés** par ce site. Cette table référence l'ensemble des segments appartenant aux partitions localisés par le site, et chaque référence contient la localisation exacte du segment référencé (c'est-à-dire le site primaire et l'emplacement du descripteur de segment sur ce site).

- Un site donné est le site primaire d'un certain nombre de segments. Ces segments sont référencés dans une table.

- Chaque site qui utilise un segment mais qui n'en est pas le site primaire possède un **cache de descripteur**. Ce cache contient entre autres la localisation du descripteur primaire.

- Enfin, il existe une table d'**accélérateurs de localisation** qui indiquent la localisation de certains segments récemment utilisés, de manière à accélérer la recherche, c'est-à-dire éviter de passer par le site de localisation.

Le descripteur de segment peut être déplacé pour des raisons d'efficacité ou d'administration. Dans ce cas, le site de localisation est notifié, de même que tous les sites possédant un accélérateur de localisation ou un cache de descripteur sur ce segment. Ainsi, un accélérateur, s'il existe, est toujours valide. Cette mise à jour systématique est utilisable parce que le déplacement d'un descripteur est un événement rare relativement au coût des notifications.

La partition de l'espace virtuel est relativement statique. Il n'y a lieu de déplacer des partitions que lors de l'ajout ou du retrait d'une machine. De plus, on n'aura besoin de créer de nouvelles partitions que lorsqu'une nouvelle machine est ajoutée au réseau ou lorsqu'une partition arrive à épuisement. De fait, la gestion des partitions peut être réalisée par un service réparti, fonctionnant en dehors du système. Elle peut même être faite à la main par l'administrateur du système.

3.3.3 Allocation des segments

Lors de la création d'un segment, il est nécessaire de choisir soigneusement l'adresse à laquelle le segment est créé, puisque le segment ne pourra pas changer d'adresse de base par la suite, et que cette adresse de base sert d'identificateur pour le segment, notamment pour l'identification du site de localisation.

Lors de la demande de création d'un segment, l'application peut, optionnellement, spécifier dans quelle partition elle veut créer le segment. Si une telle contrainte

n'est pas exprimée, le système choisit une partition selon certains critères de charge sur les sites de localisation.

Considérons qu'il faut créer un segment de taille T dans la partition P. Si le site de localisation de la partition est le site local, la solution est simple : il suffit d'avoir un pointeur de plage libre dans P, allouer le segment à cette adresse, et incrémenter le pointeur de T. Par contre, si le site localisateur de P est distant, il faudrait un échange de messages avec ce site.

Pour éviter cela, nous utilisons une allocation à deux niveaux : au niveau de la partition, nous allouons des **blocs d'allocation** ; puis, dans ces blocs, nous allouons les segments. Les blocs d'allocation sont des plages d'adresses de taille intermédiaire entre celle des partitions et celle des segments. Chaque site possède un bloc d'allocation appartenant à chacune des partitions existantes dans le système, et alloue les segments dans les blocs correspondant aux partitions dans lesquelles il faut les allouer, et ceci sans avoir à contacter le site localisateur des partitions concernées. Lorsqu'un bloc est épuisé, il est remis au site localisateur de la partition à laquelle le bloc appartient, et un nouveau bloc est retiré.

De plus, chaque site possède un bloc courant dans lequel il alloue les segments pour lesquels aucune partition n'a été demandée. Ce bloc est pris dans une partition choisie en fonction de la charge de son site localisateur. Un nouveau choix est fait à chaque retrait de bloc.

De cette manière, on réduit les communications entre le site qui veut créer un segment et le site localisateur de la partition dans laquelle il faut le créer. Cela permet une création très rapide de segments, et justifie l'utilisation d'Arias pour des données dont la durée de vie est très courte.

3.4 La cohérence et la synchronisation

Une critique souvent adressée aux systèmes de mémoire virtuelle partagée est qu'ils font payer à toutes les applications le coût d'un service de cohérence et de synchronisation qu'elles n'ont pas demandé et qui s'avère souvent mal adapté à leurs besoins. Après avoir étudié les différents modèles de cohérence et de synchronisation (désormais CeS) proposés dans la littérature, nous concluons qu'aucun modèle de CeS n'est idéal. La prolifération de modèles ne fait que soutenir cette conclusion.

Dans Arias aucun modèle de cohérence et de synchronisation n'est imposé aux applications. Celles-ci ont la possibilité de choisir parmi un ensemble de modèles celui qui leur convient le mieux. Toutefois, elles ne sont pas contraintes à le faire. Si une application n'a pas besoin de synchroniser ses accès à la mémoire, elle n'utilise aucun modèle et ne paie aucun surcoût. Arias offre la possibilité d'ajouter de nouveaux modèles de cohérence et de synchronisation à l'ensemble de base.

Dans cette section nous présentons notre approche et sa justification, ensuite nous décrivons l'architecture de notre service de CeS et les fonctionnalités qu'il assure. En fin de section la mise en œuvre du protocole de cohérence à l'acquisition illustre l'utilisation de notre service.

3.4.1 L'approche Arias

Nous prenons une approche par factorisation. Nous proposons la construction d'un support (une couche générique) qui prend en charge les fonctionnalités d'un service de CeS qui ne limitent pas le choix du modèle. Le service de cohérence et de synchronisation sera complété par un protocole spécialisé de CeS qui s'appuie sur la couche générique et qui met en œuvre un modèle particulier.

Après avoir passé en revue les tâches à la charge d'un service de CeS, nous sommes arrivés à la conclusion que les seules tâches qui ne limitent pas le choix du modèle sont la définition de l'unité de cohérence et de synchronisation, et la localisation de ces unités dans le système.

L'unité de cohérence et synchronisation que nous proposons est la zone. Une *zone* est une suite d'octets dépourvue de toute sémantique et sans limite de taille mais totalement incluse dans un segment (cf. section 3.3). Une zone est désignée par son adresse initiale et spécifiée par son adresse et sa taille.

Toute zone a une copie dite copie maîtresse qui réside dans un site particulier appelé le *maître* de la zone. Le site maître est responsable de décider de tous les aspects de synchronisation et de cohérence concernant la zone (nombre de copies, mobilité, protocole de synchronisation, etc.). Toute zone présente dans le système a un site maître à tout moment, mais le site maître d'une zone ne reste pas forcément toujours le même. La localisation des zones dans le système est basée sur le concept de maître de zone.

Si une application a besoin d'accéder à une zone, elle en fait la demande auprès d'un protocole de CeS spécifique ; si ce protocole s'aperçoit que le maître de la zone est le site même, il prend en charge la demande. Dans le cas contraire, elle est dirigée vers le maître sans que le protocole spécifique ait à s'inquiéter de la recherche du maître de la zone. A un instant donné une zone est associée à un seul protocole spécifique de CeS, mais cette association n'est pas irrévocable.

La fonction principale de la couche générique de cohérence et synchronisation est de fournir une association *zone / maître de zone* transparente aux protocoles spécialisés. Nous avons décidé que ces services prendraient l'allure d'une couche d'envoi et de réception de messages. La couche générique permet donc l'envoi de messages vers le maître d'une zone déterminée et la réception de la réponse.

Pour répondre aux besoins d'un outil de diffusion sélective dans plusieurs protocoles (par exemple. copies multiples avec invalidation), nous fournissons une abstraction supplémentaire : le *bus*. Un bus permet d'associer plusieurs messages et d'attendre toutes ou une partie des réponses pour continuer.

Les éléments de la couche générique et son architecture sont détaillés dans la section suivante.

3.4.2 Architecture générale du service de CeS

Sur tous les sites nous allons avoir la couche générique de synchronisation et de cohérence, et sur cette couche, les protocoles de CeS spécialisés. Les applications peuvent ou non utiliser un protocole spécialisé pour faire leurs accès à la mémoire. La figure Fig. 3.2 montre cette situation.

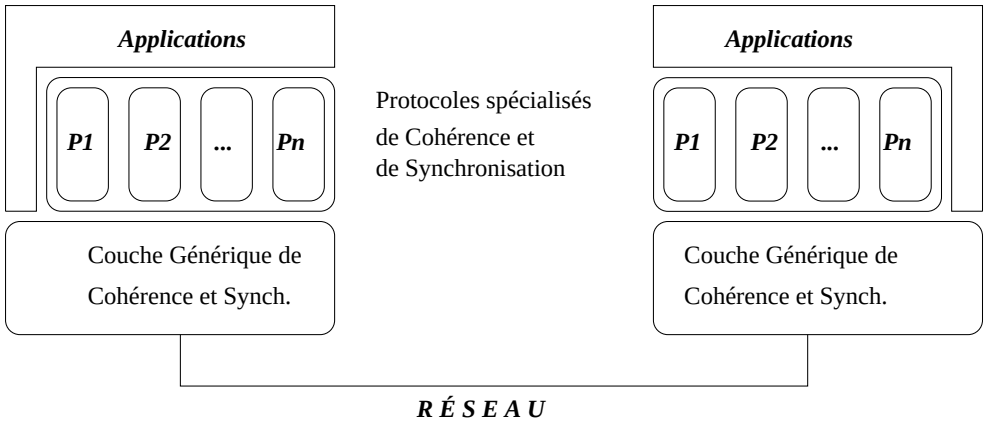

FIG. 3.2 – *Architecture générale du sous-système de cohérence et synchronisation*

La couche générique va acheminer les messages de synchronisation et de cohérence au maître de la zone concernée et leurs réponses. Ces messages contiennent deux types d'informations :

- Des informations de synchronisation et de cohérence à interpréter par le protocole spécialisé. Ces informations consistent, par exemple, en demandes/accords des droits d'accès ainsi qu'au contenu des zones. Une zone est envoyée soit pour répondre à une demande soit pour maintenir la cohérence entre les différentes copies.

- Des informations qui concernent exclusivement la couche générique. Ces informations indiquent la zone concernée par le message, le protocole spécialisé mis en jeu et des informations concernant la nature du message (déplacement de la maîtrise de la zone, envoi d'une copie de la zone, ou réponse à un message).

Pour atteindre le maître d'une zone, la couche générique fait appel au service de localisation (cf. section 3.3) qui lui indique qui est le site primaire du segment auquel la zone appartient. Sur le site primaire le *descripteur du segment* stocke l'association zone maître de zone qui permet à la couche générique de savoir à quel site elle doit envoyer le message. Des caches sur tous les sites nous permettent de stocker l'identité des maîtres des zones auxquelles on a accédées auparavant. Ces caches sont des fragments des descripteurs des segments qu'on a couplés. Ils sont mis à jour lors d'un défaut de zone dans le cache.

Un protocole de cohérence est un ensemble de routines et de structures de données placées sur tous les sites et identifiées de manière unique. Chaque protocole spécialisé doit avoir une routine de gestion de messages (un *Handler*) associée. Cette routine est activée lorsqu'un message pour ce protocole est reçu par la couche générique. Le *Handler* est chargé d'interpréter les messages qui arrivent au protocole et de déclencher les actions nécessaires.

La couche générique fournit le moyen d'enregistrer de nouveaux protocoles. Elle leur associe des identificateurs uniques et stocke l'association routine identificateur de protocole.

3.5 La permanence

La mémoire partagée répartie est composée de segments. Un segment est un ensemble de pages contiguës. Certains de ces segments peuvent être rendus résistant aux pannes. Cette résistance aux pannes est obtenue en gérant une image permanente cohérente de chaque segment. Cela signifie qu'ils vont posséder une image stable sur un support tel qu'un disque. Dans cette section, nous allons étudier de quelle manière l'image d'un segment permanent en mémoire d'exécution et celle en mémoire permanente interagissent.

Le but que nous poursuivons pour la mémoire permanente est de fournir un ensemble de services permettant de construire différentes politiques de stockage pour les segments. L'intérêt d'avoir plusieurs politiques réside dans ce qu'elles peuvent fournir différents niveaux de fiabilité et de performance pour une classe d'applications donnée.

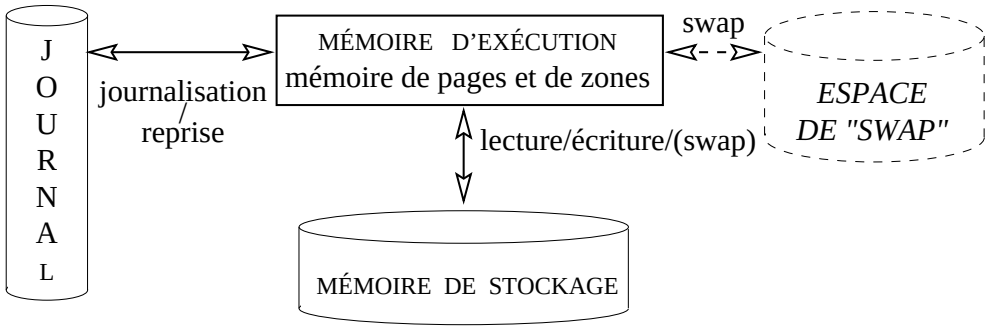

FIG. 3.3 – *Les différents éléments du service de stockage*

La figure Fig. 3.3 nous montre les différents éléments qui jouent un rôle dans le service de stockage. Cette figure donne une vision purement centralisée du service de stockage ; la répartition des différents éléments sera traitée dans la suite. Notre système gère trois espaces de stockage (sur disque) qui sont disjoints et indépendants :

- la mémoire de stockage,
- le journal,
- l'espace de pagination.

Nous allons détailler dans la suite le rôle de chacun de ces éléments. Nous allons voir que la gestion de la mémoire d'exécution a besoin d'interagir avec ces trois entités. Deux aspects interviennent dans la gestion de la mémoire d'exécution : la gestion de la pagination (gestion des blocs de mémoire centrale) et la gestion de la couche générique de cohérence (gestion des zones au-dessus des pages). La gestion de l'espace permanent est transparente aux applications ; c'est donc au travers des deux couches citées précédemment que la gestion du stockage va pouvoir être paramétrée de manière à permettre différentes politiques.

3.5.1 La mémoire de stockage

La mémoire de stockage contient l'ensemble des images permanentes des segments. Elle est composée des mémoires de stockage de chacun des sites participant à la mise en œuvre du système. Chacune d'elles est divisée en volumes qui vont contenir les images résistantes des segments. Une image résistante de segment est entièrement contenue dans un volume. Pour assurer la disponibilité des segments permanents en cas de panne du site qui les contient, les volumes devraient être dupliqués. L'étude de la duplication des volumes ne fait cependant pas partie de la première phase du projet et dépasse le cadre de ce document. Les manipulations qui peuvent être faites sur la mémoire de stockage d'un site sont très restreintes. Elles se limitent aux fonctions suivantes :

- Création et destruction de volumes et d'images disque d'un segment permanent.
- Lecture et écriture d'une page (ou d'une partie de page) d'un segment.
- Mise en indisponibilité d'un segment permanent (le segment ne peut plus être couplé dans la mémoire virtuelle).

Les seules propriétés garanties par la mémoire permanente sont l'atomicité de la création ou de la destruction de l'image résistante d'un segment, ainsi que l'atomicité d'écriture d'une page d'un segment. En revanche, l'atomicité de mise à jour d'un segment par rapport à un ensemble de modifications n'est pas assurée. Si une application requiert une telle propriété des mises à jour qu'elle effectue, elle devra la mettre en œuvre à l'aide du service de journalisation décrit dans la sous-section 3.5.3.

En principe, nous gérons deux niveaux de mémoire indépendants : la mémoire virtuelle partagée de segments persistants et la mémoire de stockage. Il ne devrait donc pas y avoir a priori de coopération entre le gérant de chacun de ces niveaux de mémoire. Cependant, pour des raisons d'efficacité et d'économie de l'espace de *pagination*, le gérant de pagination s'appuie sur le gérant de l'espace de stockage pour la gestion des pages des segments permanents. La description de cette coopération fait l'objet de la sous-section 3.5.2.

Le service de gestion de l'espace de stockage est utilisé par les applications pour rendre un segment permanent ou pour recopier de façon synchrone des morceaux de segments permanents modifiés. Le service de stockage est aussi utilisé par les autres composants de notre système : Le recopieur asynchrone l'utilise pour recopier en mémoire permanente les zones des segments permanents dont les modifications ont été validées. La couche de cohérence et de synchronisation l'utilise pour recharger les données résistantes après une panne.

3.5.2 Utilisation de l'espace de stockage par les paginateurs

La mise en œuvre de la mémoire virtuelle partagée de segments s'appuie sur l'ensemble des mémoires centrales des stations de travail qui l'utilisent et des espaces

de *pagination* des disques attachés à ces stations. Le but poursuivi est d'éviter, chaque fois que cela est possible, deux écueils :

- La multiplication des entrées/sorties sur disques. Quand une page d'un segment permanent doit être recopiée dans l'espace de stockage, il faut éviter d'avoir à aller la chercher dans l'espace de *pagination*.

- La duplication dans l'espace de pagination d'une partie de la mémoire de stockage. Quand un bloc de mémoire centrale d'une machine doit être rendu disponible, il faut éviter de recopier systématiquement la page qui l'occupe dans l'espace de pagination associé à cette machine.

Si la page qui doit être retirée de la mémoire est une page appartenant à un segment temporaire, le paginateur ne doit la recopier dans l'espace de pagination que si son image dans l'espace de pagination doit être mise à jour. En particulier, une page qui n'a jamais été copiée dans le pagination doit l'être.

Si une page qui doit être recopiée dans l'espace de pagination est une page appartenant à un segment permanent, elle ne doit être recopiée dans l'espace de pagination que si sa valeur est différente de celle stockée dans l'espace de stockage. Avant de recopier la page dans l'espace de pagination, le paginateur recopie dans l'espace de stockage l'ensemble des zones de la page qui sont validées. On évite ainsi au recopieur asynchrone de provoquer des fautes de pages lors de la recopie en mémoire de stockage des zones validées. Si la valeur de la page est identique soit à son image en espace de stockage, soit à son image dans l'espace de pagination (la page est dite « propre »), elle n'est pas recopiée.

3.5.3 Le journal

Pour assurer, en cas de panne, l'atomicité d'un ensemble de modifications dans des segments permanents, le système fournit un service de journalisation. Ce service permet à une application d'enregistrer dans un journal des informations sur toutes les modifications qu'elle effectue de manière à permettre la réparation des segments permanents suite à une panne logicielle ou matérielle. La notion de journal telle que nous la présentons ici correspond à ce que nous nommons journal logique.

Le service de journalisation est extensible dans la mesure où il permet d'enregistrer plusieurs protocoles de reprise. Ces protocoles sont utilisés lors d'une panne d'un site pour réparer la partie de la mémoire permanente gérée par ce site. Lors de la reprise, le service de journalisation est capable de déterminer si un journal logique a été validé ou non. Il applique alors, suivant le cas, les procédures de reprise adéquates d'un protocole particulier, sachant que les enregistrements d'un journal logique sont liés à un de ces protocoles.

Un journal logique est un ensemble d'enregistrements ordonnés, tous liés à une application (transactionnelle ou non). A priori, ces enregistrements sont écrits séquentiellement par une application, l'ordre des modifications déterminant l'ordre des enregistrements dans le journal logique. Si une application met en œuvre plusieurs flots d'exécution parallèles, le service de journalisation ne garantit aucun ordre entre les enregistrements écrits par les différents flots.

Plusieurs journaux logiques différents peuvent décrire des modifications portant sur un même segment. Par exemple, cela peut arriver si deux applications modifient successivement la même zone ou des zones différentes d'un même segment. Dans le cas où les modifications portent sur la même zone, il faut que l'état final du segment corresponde à son état après les dernières modifications qui ont été effectuées.

Les actions associées aux journaux logiques sont les suivantes :

- Création d'un journal logique,
- Écriture d'enregistrements dans un journal logique,
- Préparation d'un journal logique en vue de sa validation,
- Validation d'un journal logique,
- Annulation d'un journal logique.

Il faut noter que les enregistrements contenus dans un journal logique concernent toujours des modifications opérées sur des segments ayant une image en espace permanent.

Un journal logique contient un ensemble d'enregistrements fourni par une application. De tels enregistrements sont composés de deux parties (voir figure 3.4) : un en-tête dont la structure est connue par le service de journalisation et des informations destinées à un protocole de reprise particulier.

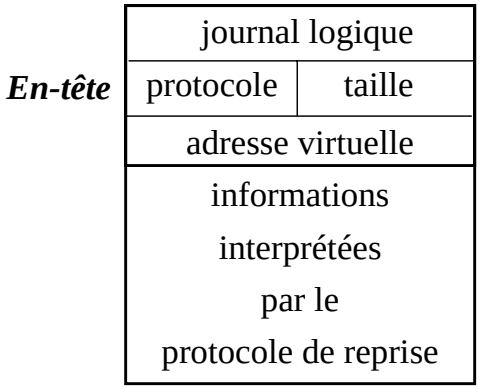

FIG. 3.4 – *En-tête d'un enregistrement de journal*

L'en-tête est composé de l'identificateur du protocole, de la taille de l'enregistrement et de l'adresse virtuelle de la zone concernée par la modification. Il est suivi par un paquet de taille quelconque d'informations définissant le type de modification. La structure et la sémantique de ces dernières informations ne sont pas connues du service de journalisation. Le rôle de ces informations est d'être traitées en cas de reprise, le service de journalisation s'occupant d'y appliquer le bon protocole de reprise. Le processus de reprise est mis en œuvre par un démon appelant de manière ascendante un certain nombre de routines prédéfinies qui sont fournies par chaque protocole.

Il est possible d'envisager deux modèles d'implantation des journaux logiques, ces modèles influant sur les performances de la journalisation et la disponibilité des segments permanents lors d'une panne. Nous présentons brièvement les modèles ci-dessous :

3.5.3.1 Modèle centralisé

Le premier modèle consiste à centraliser un journal logique sur un site unique (par exemple sur un site où s'exécute l'application) ; ce modèle garantit l'ordonnancement attendu lors d'une reprise après panne (cette solution assure un ordre total).

3.5.3.2 Modèle réparti

Le deuxième modèle consiste à répartir le journal logique. Un tel journal étant maintenant partitionné, il devient nécessaire de définir les conditions permettant de garantir l'ordonnancement. La solution consiste à ce que les partitions composant le journal logique soient indépendantes les unes des autres, c'est à dire qu'elles puissent être rejouées de façon autonome (ordre partiel).

Nous avons choisi de mettre en œuvre le second modèle. Un enregistrement du journal logique est écrit dans la partition de journal présente sur le site possédant l'image permanente du segment contenant cet enregistrement. Ce modèle nous paraît préférable car il nous permet de placer les enregistrements de journal décrivant des modifications sur un segment donné sur le même site que l'image stable de ce segment. Donc l'information nécessaire à récupérer l'image stable d'un segment réside sur le même site que cette image stable elle-même. Ainsi nous pouvons faire une récupération *indépendante* sur les différents sites.

Le service de journalisation doit être très efficace. Or la principale source de coût pour ce service est la gestion des E/S (majoritairement des écritures en phase de fonctionnement normal). Il est clair que le fait d'écrire séquentiellement sur le disque diminue le coût des écritures puisqu'il limite les mouvements de la tête du disque. La solution que nous préconisons est donc d'implanter les journaux logiques au-dessus d'un journal physique. Ce journal physique est alors matérialisé soit sur disque autonome si nous voulons garantir le maximum d'efficacité, soit dans une partition spéciale d'un disque à usage général. Chaque site participant à la mise en œuvre de la mémoire permanente possède alors un journal physique.

De plus, un service de recopie asynchrone est mis en œuvre pour que les modifications opérées des images en mémoire d'exécution des segments permanents puissent être régulièrement reportées sur l'image permanente sans pour autant pénaliser les performances lors du déroulement des transactions. En effet, il est intéressant pour différentes implantations de transactions de pouvoir recopier de façon asynchrone des modifications, sachant que le système dispose du journal pour réparer la mémoire permanente. Par ailleurs, un système de consommation du journal physique est couplé à celui de recopie asynchrone. Cela permet ainsi d'éliminer les journaux logiques obsolètes du journal physique pour y récupérer de l'espace disque.

3.6 Architecture des services d'Arias

Cette section décrit l'architecture des services décrits ci-dessus. Nous donnons une vision d'ensemble de cette architecture, c'est à dire comment se placent les différents services les uns par rapport aux autres, et de quelle manière ils interagissent. Nous distinguons deux niveaux du point de vue architectural : l'architecture logicielle et l'architecture système. Les deux sections qui suivent (section 3.6.1 et section 3.6.2) leur sont respectivement consacrées.

3.6.1 Architecture logicielle

L'architecture logicielle que nous décrivons dans cette section se place à un niveau de granularité important. Nous énumérons les principaux modules intervenant dans l'implantation. Ces modules sont représentés par des cadres en traits épais dans la figure Fig. 3.5. Nous constatons que leur code réside pour l'essentiel dans le noyau du système Unix. La section 3.6.2 nous explique d'ailleurs quels sont les mécanismes et services du noyau qui sont utilisés.

Sur cette même figure, on trouve au plus haut niveau les applications, tandis qu'au-dessous se trouvent des modules spécialisés (traits fins) qui s'appuient directement sur les services génériques (trait épais). Ce niveau démontre l'extensibilité du système ; nous constatons d'ailleurs que ces modules résident pour leur plus grande partie dans le noyau, offrant ainsi une protection maximale (vis-à-vis des applications) des structures d'exécution nécessaires à ces modules.

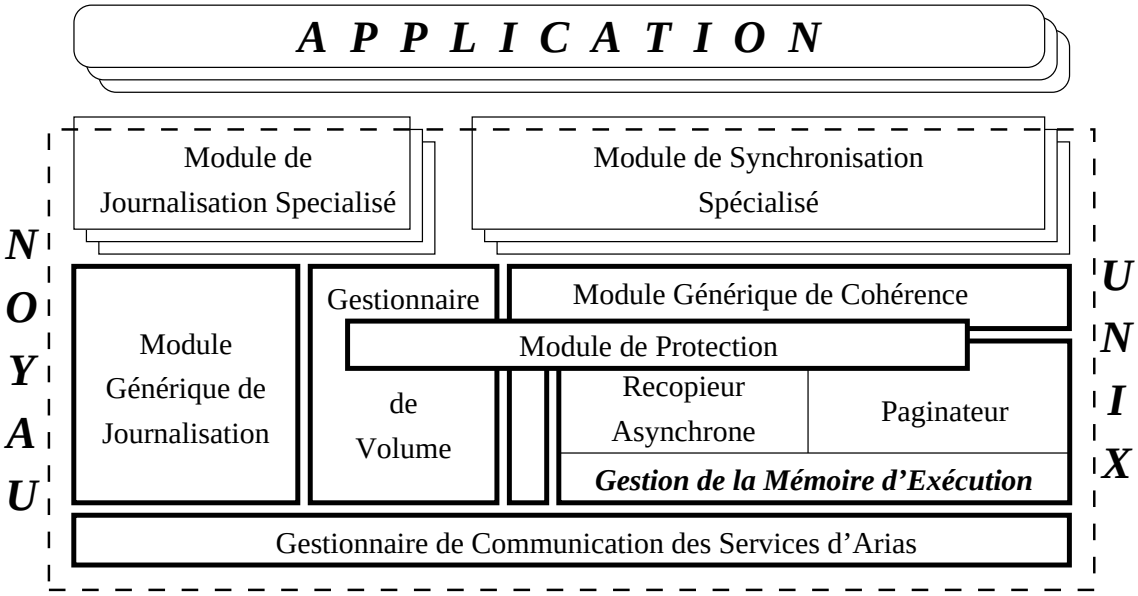

FIG. 3.5 – *Les modules logiciels*

Nous allons maintenant détailler chacun des modules mis en œuvre pour l'implantation des services génériques (traits épais). Nous commençons par les modules et sous-modules liés à la gestion de la mémoire d'exécution. Ils sont au nombre de

cinq :

1. Celui de plus haut niveau est le Module Générique de Cohérence. Il offre l'interface de manipulation des zones pour les Modules de Synchronisation Spécialisés. C'est à travers lui que ces modules vont pouvoir paramétrer le comportement du gestionnaire de la mémoire d'exécution. Il offre en plus un moyen de dialoguer entre des protocoles en dehors de la gestion des zones. Il s'appuie donc sur le gestionnaire de la mémoire d'exécution, ce dernier étant composé de trois sous-modules.

2. Le premier sous-module est le paginateur. Il s'agit d'un module qui est activé par le matériel pour traiter les défauts de page (exécution locale à un site). Il s'occupe donc de matérialiser des pages nouvelles en mémoire centrale ou de les vider de cette même mémoire (gestion des blocs de mémoire physique). Les traitements effectués lors d'un vidage utilisent les informations liées aux zones gérées dans la page concernée. Par ailleurs, le paginateur participe aussi au processus de recopie asynchrone.

3. Le deuxième sous-module est le gestionnaire de zones. Il enregistre toutes les zones qui ont été demandées sur un site ainsi que leur statut. Il interagit au niveau de la gestion de la mémoire d'exécution avec la gestion des pages (c'est-à-dire le paginateur) ainsi qu'avec le gestionnaire des segments (couplage, localisation, etc...) qui n'apparaît pas dans la figure Fig. 3.5.

4. Le troisième sous-module est le recopieur asynchrone. Son rôle est essentiellement de vider des zones de segments permanents dont l'image en mémoire d'exécution a été déclarée comme « copiable » vers leur image en mémoire permanente. Ce module interagit donc avec le module de stockage pour les recopies ainsi qu'avec le module générique de journalisation pour la consommation du journal.

5. Le dernier module impliqué dans la gestion de la mémoire d'exécution est le module de protection. Il intervient exclusivement lors du couplage d'un segment dans l'espace d'adressage d'un processus Unix. Tout processus Unix utilisant les services s'exécute dans un domaine de protection. Le module s'occupe donc de vérifier si les conditions de couplage d'un segment sont remplies par rapport au domaine dans lequel le processus s'exécute. Ce module doit aussi interagir avec le module de stockage et peut-être de journalisation car la définition des domaines doit pour une grande partie être résistante, c'est à dire permanente.

En plus de tous ces modules se rapportant à la gestion de la mémoire d'exécution, nous avons deux modules plus ou moins indépendants qui offrent des services connexes :

1. Le module de stockage s'occupe de la gestion de la mémoire permanente. Il gère les volumes et les segments permanents à l'intérieur de ceux-ci. Le niveau de service offert par ce module est très proche de ce qu'offre un système de

fichiers, où les fichiers seraient les segments permanents dont les noms seraient des adresses virtuelles.

2. Le module générique de journalisation gère la journalisation de modifications opérées sur des segments permanents. Il permet de supporter différents types de journaux logiques avec leur protocole de reprise associé. Ces journaux logiques sont physiquement multiplexés dans un journal physique, garantissant un fort débit d'écriture. Il supporte un mécanisme de consommation du journal physique basée sur la déclaration d'obsolescence de journaux logiques.

Enfin, la plupart de ces modules offrant un service réparti, ils s'appuient au plus bas niveau sur le gestionnaire de communication. Le service fournit par ce module est du type envoi de message et gestion des réponses.

Comme nous allons le voir dans la section suivante (sous-section 3.6.2), ces modules sont inclus dans le noyau Unix sous la forme d'extensions. Cela est aussi vrai en partie pour les modules spécialisés de synchronisation ou de journalisation. Ces modules offrent une interface aux applications, cette interface étant mise à leur disposition sous la forme d'une librairie liée au code des applications.

3.6.2 Mise en œuvre sur un système Unix

Le but de cette section est de donner une première idée de la manière dont les services vont être introduits dans le système Unix. Notre idée maîtresse est d'utiliser au maximum le mécanisme de «stream» du système Unix. Pour résumer, un «stream» est un canal de communication bidirectionnel. Ce canal est ponctué d'un certain nombre de modules (modules «stream») qui vont agir, chacun à son tour, comme des filtres sur les messages qui transitent sur le canal, dans un sens comme dans l'autre. Au plus bas niveau, le canal aboutit généralement à un mécanisme matériel représenté par un pilote («stream driver»). Ce mécanisme présente de nombreux avantages, notamment la modularité, l'efficacité grâce à un comportement asynchrone (interactions non bloquantes pour le processus utilisateur) et la relative facilité de mise en œuvre. Par ailleurs, il s'agit d'un mécanisme standard ce qui nous offre des garanties de portage.

La figure 3.6 nous montre l'organisation possible des pilotes et modules implantant les services. Au plus bas niveau, nous observons le pilote («pseudo driver») «stream» implantant le module de communication utilisée par la plupart des services. Ce pilote s'appuie ici directement sur les sockets Unix (utilisation de TCP/IP). Nous pourrions néanmoins envisager de remplacer ce pilote par un autre dans le cas où nous voudrions nous placer sur un autre type de support de communication comme par exemple du «Fiber Channel» ou de l'ATM à un niveau bas (hors TCP ou UDP).

Au-dessus de ce pilote, nous avons trois modules «stream» correspondant au trois services manipulés par les applications à travers les modules spécialisés. Il s'agit du service de stockage et des services génériques de journalisation et de gestion des zones (gestion de la cohérence mémoire). Ces trois modules sont empilés les uns sur les autres et correspondent à autant de filtres pour les messages issus des modules applicatifs. Un tel message peut donc contenir des informations pour les

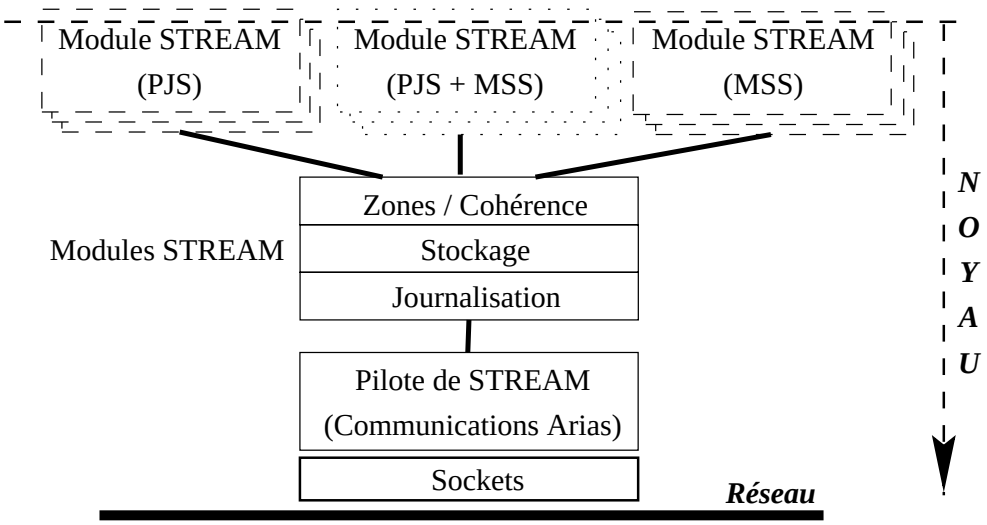

FIG. 3.6 – *Implantation des services à base de « stream »*

trois couches simultanément (ou pour une seule ou pour deux d'entre elles), ce qui permet d'économiser le nombre de messages à traiter et à envoyer.

Le module de protection n'est pas visible dans la figure 3.6 ; il s'agit d'un module relativement indépendant qui va fournir une interface directe pour les applications nécessitant de manipuler explicitement le service de protection.

Enfin, les modules «stream» de la couche supérieure sont déjà des modules applicatifs vis-à-vis des services. Il s'agit de modules «stream» implantant soit des Protocoles de Journalisation Spécialisés (PJS), soit des Modules de Synchronisation Spécialisés (MSS). Nous pouvons observer que pour les MSS, il peut y avoir des interactions à la fois avec le module de gestion des zones et le module de stockage. Par ailleurs, pour des raisons d'efficacité, un MSS et un MJS peuvent être implantés dans un seul module «stream» et utiliser tous les modules de la couche inférieure.

La figure Fig. 3.6 ne montre pas tous les éléments intégrés dans le noyau. C'est notamment le cas pour les entités actives tels que des processus «noyau». Nous allons avoir ce type de processus sur chaque site, notamment pour le recopieur asynchrone ou pour la gestion des E/S disques asynchrones. Cette section ne donne donc qu'une ébauche de l'architecture système d'implantation des services. Nous insistons sur le fait que la plupart des mécanismes utilisés sont des standards, un de nos objectifs prioritaires étant la portabilité.

3.7 Validation

Comme cela a été souligné auparavant, l'intégration avec un système Unix est un aspect important du projet. Nous avons choisi de réaliser cette plate-forme sur le système AIX, le système Unix fourni par la compagnie Bull, pour les raisons suivantes :

– Le projet Sirac fait suite au projet Guide (mené par la même équipe), qui a débouché sur le développement d'un prototype industriel appelé OODE sur

le système AIX. OODE est un environnement permettant le développement d'applications réparties à l'aide d'objets partagés persistants. Les applications sont écrites à l'aide d'une extension du langage C++. La gestion du partage et de la persistance sont à la charge du système, ce qui facilite grandement la programmation.

Notre objectif est donc de fournir un support bien plus adapté qu'Unix ou que des micro-noyaux tels que Mach ou Chorus pour un un environnement tel que OODE.

- AIX est le système Unix distribué par la compagnie Bull, qui est représentée dans le projet Sirac. Nous pouvons ainsi profiter de l'expertise des ingénieurs de Bull.

- Le système AIX semble bien adapté à l'accueil d'une telle extension, comme cela a été souligne dans la section 3.6.

Le développement doit être suivi d'une validation dans les conditions les plus réalistes possibles. De plus, nous considérons qu'une recherche en systèmes doit déboucher sur la réalisation de prototypes utilisables, comme cela a été le cas pour le système Guide. Dans le cadre de Sirac, nous nous intéressons plus particulièrement au support des environnements suivants :

- Le support d'environnements répartis orienté objets.

  Comme cela a été mentionné ci-dessus, notre équipe s'est précédemment intéressée à ces environnements de développement répartis dont OODE est un exemple. Ces environnements font une utilisation intensive de données partagées et persistantes. Une des validations de notre travail consiste à porter l'environnement OODE sur nos services afin de montrer que notre support est plus adapté que l'utilisation directe des fonctions d'AIX.

  De plus, le support de l'environnements OODE nous permettra de récupérer les applications développées et en cours de développement à des fins d'évaluation en grandeur réelle.

- Le support de bases de données.

  Nous visons également le support efficace de systèmes de gestion de bases de données. Ces systèmes, qui sont souvent réalisés sur Unix, doivent généralement redévelopper des services pour la persistance et le partage des données. Nous proposons de réutiliser les services que nous avons décrits.

- Le support de programmes parallèles.

  Nous pensons que notre plate-forme sera également adaptée au support de programmes parallèles. Nous nous intéressons au support d'un environnement comme PVM ainsi qu'au support pour la parallélisation de serveurs haute performance, par exemple pour des serveurs Web soumis à une charge importante.

– Réalisation différente de services existants.

  Une autre validation possible consiste à réaliser des services systèmes existants en utilisant nos fonctions de partage et de permanence. Il sera notamment intéressant de réaliser un système de gestion de fichier réparti à partir de ces fonctions. L'objectif sera alors de gérer ces fichiers au moins aussi efficacement qu'Unix, mais en offrant en plus les propriétés de résistance aux pannes décrites auparavant.

3.8 Comparaison avec l'existant

Dans cette section, nous situons nos travaux par rapport aux projets du même domaine. Pour cette comparaison, nous passons en revue les quatre axes de travail traités dans les sections 3.3, 3.4 et 3.5.

3.8.1 Gestion d'un espace virtuel unique

Plusieurs projets de recherche explorent actuellement l'utilisation des processeurs 64 bits pour la gestion d'un espace virtuel unique dans un système réparti. Il s'agit notamment des projet Opal [CLBHL92, CLLBH92a], Mungi [HERV93] et Angel [MSWK93]. Si l'utilisation des adresses virtuelles comme nom unique est un acquis, peu de choses ont été proposées pour l'allocation des adresses, la localisation ou la migration des segments dans cet espace. C'est un des aspects sur lesquels nous comptons apporter de nouvelles solutions.

3.8.2 Cohérence et synchronisation

Le couplage fort entre la cohérence et la synchronisation est inspiré du travail réalisé dans le projet Midway [BZ91], dans lequel la cohérence à l'entrée (*Entry Consistency*) est proposée.

La gestion de différents protocoles de maintien de la cohérence a déjà été étudiée dans des projets de recherche comme Munin [BCZ91], mais aucun système à l'heure actuelle ne permet aux applications de spécifier leur propre protocole de gestion de la cohérence. Munin laisse seulement le choix entre un nombre limité de protocoles de cohérence implantés dans le système.

3.8.3 Permanence

De même que pour la cohérence, les systèmes actuels ne permettent pas aux applications de spécialiser la gestion des journaux en fonction de besoins très particuliers. Nous fournissons aux applications la possibilité d'assurer la permanence de leurs segments en mettant en œuvre une politique optimale de journalisation.

3.8.4 Protection

Tous les systèmes fondés sur la gestion d'un espace virtuel unique fournissent la protection sous forme de capacités logicielles et de domaines de protection

(Opal [CLBHL92, CLLBH92a], Mungi [HERV93]). Toutefois, ces systèmes imposent aux utilisateurs la manipulation directe des capacités. Ainsi, le programmeur d'application a la charge d'une partie de la gestion des capacités. Nous proposons de gérer les capacités de façon transparente à l'utilisateur, ce qui rend le code des applications indépendant de la protection.

4

Conception du système de stockage des données

Après avoir fait un tour d'horizon de tous les services Arias dans le chapitre précédent, nous allons maintenant nous concentrer sur le service de stockage, et plus particulièrement sur le service de journalisation. Dans la première section nous illustrons brièvement les propriétés désirables d'un service de stockage. Nos points forts sont la souplesse et l'intégration dans un contexte général. Cette intégration avec les autres services d'Arias est décrite dans la deuxième section. Cette deuxième section donne également un aperçu sur le format de nos journaux, qui permet de découper le sous-système de journalisation en une partie générique et une partie spécifique à l'application. La troisième section décrit nos mécanismes de validation atomique et de récupération après une panne, en distinguant les deux cas de figure «journalisation avant» et «journalisation après». Dans la quatrième section nous illustrons comment notre service de stockage s'intègre avec la gestion de la mémoire virtuelle. Cette intégration nous apporte une meilleure efficacité en évitant la double pagination. La cinquième section décrit comment nous gérons la purge du journal. La dernière section donne quelques idées sur la réplication des données.

4.1 Buts du service de stockage

Cette section énumère les buts de notre service de stockage, et souligne les difficultés que nous avons rencontrés en concevant une méthode de reprise après pannes qui supporte les fonctionnalités que nous recherchons.

L'objectif du service de stockage est d'assurer la permanence des données, même après une panne. Dans Arias, un segment de la *mémoire virtuelle partagée persistante et distribuée* peut être rendu *permanent*, c'est-à-dire recevoir une image sur un support permanent, (disque, mémoire vive non volatile, etc.) ce qui lui permet de résister aux pannes du système.

4.1.1 Les propriétés *ACID*

Le concept de transaction comprend les propriétés *ACID (Atomicité, Cohérence, Isolation et Durabilité)*[HR83]. Dans Arias, les propriétés de cohérence et d'isolation

des transactions sont garanties par le gestionnaire de la cohérence et de la synchronisation. Le service de stockage doit uniquement garantir la durabilité et l'atomicité. Deux mécanismes sont fournis pour la gestion de la permanence : la durabilité à long terme est assurée par le gestionnaire des volumes qui gère *l'image stable* des segments. Il permet de rendre permanente l'image d'un segment (le segment passe alors de l'état volatil à l'état permanent). L'atomicité, et aussi la durabilité *à court terme*, sont assurées par les serveurs de journalisation, qui utilisent un *Write Ahead Log (WAL)*. Il s'agit là d'un journal dans lequel on écrit avant de mettre à jour l'image stable. L'utilisateur peut faire enregistrer une liste de modifications dans le journal, puis, après validation du journal, répercuter ces modifications sur l'image permanente de la mémoire. En cas de panne, le journal permet de reconstruire une image cohérente des données. Les modifications validées sont retenues, tandis que les modifications non validées dans le journal sont perdues.

Notre système supporte aussi des applications non transactionnelles. Ceci est utile dans des situations où il faut moins de fiabilité mais plus de rapidité. Ces applications ne bénéficient pas nécessairement des propriétés C et I, mais elles gardent quand même A et D : toutes les données sauvegardées lors d'un même point de reprise sont sauvegardées de manière atomique et durables [Gra78].

L'indépendance et la cohérence sont assurées par le gestionnaire de cohérence, qui ne fait pas partie du système de stockage dans Arias.

4.1.2 Répartition

Comme notre système est réparti, nous utilisons un *protocole de validation à deux phases* pour nous assurer que les différents serveurs de stockage sont d'accord au sujet de l'état des transactions. Nous avons fait des efforts pour rendre possible une *reprise partielle* pour les transactions réparties, même dans des cas où tous les sites participants ne sont pas disponibles.

4.1.3 Souplesse

Contrairement à la plupart des systèmes transactionnels actuels, Arias permet aux applications de décider elles-mêmes comment assurer la cohérence entre la mémoire d'exécution, le journal, et l'image stable. Une application peut donc choisir la politique de cohérence qui lui va le mieux. Par exemple, elle peut utiliser un *journal avant* et défaire les modifications en cas d'abandon, ou bien elle peut utiliser un *journal après*, et répercuter les modifications sur l'image lors de la validation[15]. Plusieurs protocoles de journalisation sont disponibles sous forme de librairies et de modules noyau, appelés *Protocoles de Journalisation Spécifiques (PJS)*. L'application peut soit utiliser un des PJS disponibles, soit installer son propre PJS dans le noyau.

15. Le journal avant et le journal après sont tous les deux des cas particuliers du *Write Ahead Log (WAL)*. En effet, dans les deux cas, on écrit «des choses» sur le journal avant de mettre à jour l'image. La différence se situe dans la nature de «ces choses». Dans le cas du journal avant, il s'agit d'informations permettant de défaire les modifications pour obtenir l'état avant la transaction, alors que dans le cas du journal après, il s'agit d'informations permettant de refaire les modifications pour obtenir l'état après la transaction.

4.1.4 Performance et Intégration

Un de nos buts est la performance à tous les niveaux : haut débit des transactions et faible latence des validations. Afin de limiter les entrées-sorties disque, nous intégrons notre système à la mémoire virtuelle. Ceci est décrit en détail dans la section 4.4.

4.2 Intégration du service de stockage dans son contexte général

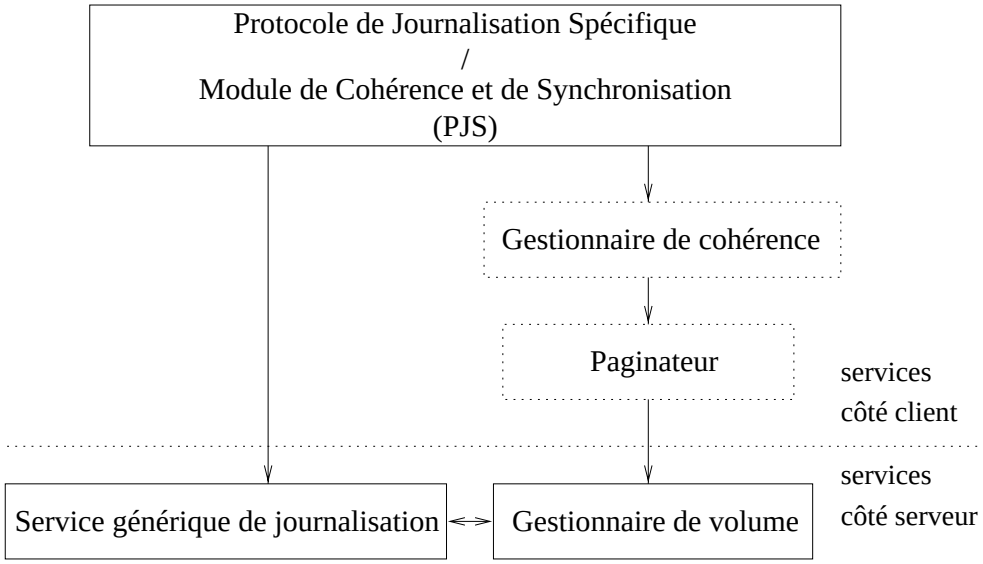

FIG. 4.1 – *Architecture du système*

La figure 4.1 décrit les différents composants de notre système. Ceci est une description fonctionnelle, et chaque boîte de la figure représente des entités qui sont distribuées sur plusieurs sites. La répartition est faite sur deux plans :

- Répartition côté client : Une application peut avoir plusieurs flots d'exécution coopérants qui se déroulent sur plusieurs sites. Les services se situant du côté de l'application (client) sont représentés au-dessus de la ligne en pointillé. Ces services se trouvent sur le même site que le flot d'exécution applicatif utilisant les objets.

- Répartition côté serveur : Il est possible que les différentes données utilisées soient basées sur des sites différents. Les services côté client sont représentés en dessous de la ligne en pointillé. Ces services se trouvent sur le même site que l'image permanente des objets manipulés.

Nous distinguons les services suivants :

- Les *Protocoles de Journalisation Spécifiques (PJS)* sont la couche la plus haute du service de journalisation. Ils assurent la cohérence entre le journal et l'image,

et prennent, pour le compte de l'application, l'initiative de journaliser et de mettre à jour l'image. Chaque PJS offre un protocole de journalisation différent, qui est adapté à une classe différente d'applications. Le PJS offre aussi un protocole de reprise qui rejoue (ou défait, selon les cas) le journal lors de la reprise après panne.

- Le *Service Générique de Journalisation (SGJ)* est responsable pour le stockage physique des enregistrements de journal sur le disque. Pendant la récupération, le service de journalisation générique prend l'initiative, et déroule le protocole d'accord.

- Le *Gestionnaire de Cohérence* est responsable de la gestion des verrous entre les applications, et de la cohérence de mémoire entre sites.

- Le *Paginateur* gère le va-et-vient. Il participe à la gestion du stockage en envoyant les données validées au gestionnaire de volume plutôt qu'au disque d'échange.

4.2.1 Souplesse et format du journal

Afin de pouvoir séparer la tâche de journalisation en deux modules (PJS et SGJ), nous subdivisons chaque enregistrement du journal en deux parties :

- Un en-tête qui contient les informations indépendantes du protocole. Cet entête peut être compris par le SGJ et par le PJS,

- Une partie « données » qui est interprétée uniquement par le PJS. Cette partie est spécifique au protocole, et décrit une modification à apporter au segment. Afin de simplifier la procédure de reprise après panne, et afin d'éviter une manipulation compliquée *d'enregistrements de compensation*, nous imposons la contrainte que les applications journalisent uniquement des opérations *idempotentes*. En pratique, cette contrainte est facile à satisfaire.

Ce format permet au SGJ d'ignorer les détails du protocole de journalisation utilisé, à l'exception de la distinction entre la journalisation *avant* et la journalisation *après*.

4.3 Récupération après une panne

Cette section décrit quelles actions doivent être effectuées après une panne afin d'assurer l'atomicité et la durabilité. Nous allons d'abord décrire la récupération dans le cadre d'un protocole qui utilise la *journalisation avant*. Nous supposons ici qu'une panne affecte uniquement l'état *volatile* d'un site : toute la mémoire d'exécution (mémoire centrale, et espace de pagination) est perdue, mais la mémoire de stockage (journal et image stable) reste intacte.

En effet, cette hypothèse est justifiée par le fait que des problèmes de logiciels et des pannes de courant sont beaucoup plus fréquents que des pannes de disque.

Pendant l'opération normale, nous écrivons d'abord vers le journal les enregistrements qui décrivent des changements, et puis nous validons le journal. Quand le journal est validé, et que tous les enregistrements sont stockés sur disque, les changements sont répercutés sur l'image stable. Toutes ces actions sont entreprises à l'initiative du PJS qui agit pour le compte de l'application.

Si une panne se produit avant que le journal ne soit validé, la transaction est abandonnée, et donc tous les changements sont écartés. Comme aucune modification n'a encore été faite sur l'image stable, aucune autre action n'est nécessaire pour abandonner la transaction.

Si au contraire la panne se produit après la validation, la transaction est considérée comme faite entièrement. Cependant, en réalité, tous les changements induits par cette transaction n'ont pas été nécessairement répercutés sur l'image stable. Donc, dans ce cas, l'image stable doit être remise en cohérence avec le journal transactionnel. Afin de faire ceci, les enregistrements de journal relatif à la transaction en question sont *rejoués*.

4.3.1 Transactions multi-sites et validation à deux phases

Chaque serveur de stockage participant à une transaction multi-sites gère son propre *Write Ahead Log*. Afin de garantir l'indépendance entre ces différents sites de stockage après une panne, nous faisons en sorte que les journaux décrivant les modifications d'un segment soient stockés sur le même site que l'image stable de ce segment. Cependant, une transaction peut affecter des segments qui sont stockés sur des sites différents. Donc le journal de la transaction est étalé sur plusieurs sites lui aussi. Nous disons que le *journal logique* de la transaction est subdivisé en plusieurs *Journaux Logiques Locaux (JLL)*, qui sont stockés dans les *Journaux Physiques* des différents serveurs. Comme la figure 4.2 le montre, l'étalement d'un journal logique sur plusieurs sites, et le rassemblement de plusieurs journaux logiques locaux au sein d'un même journal physique jouent des rôles duaux ici.

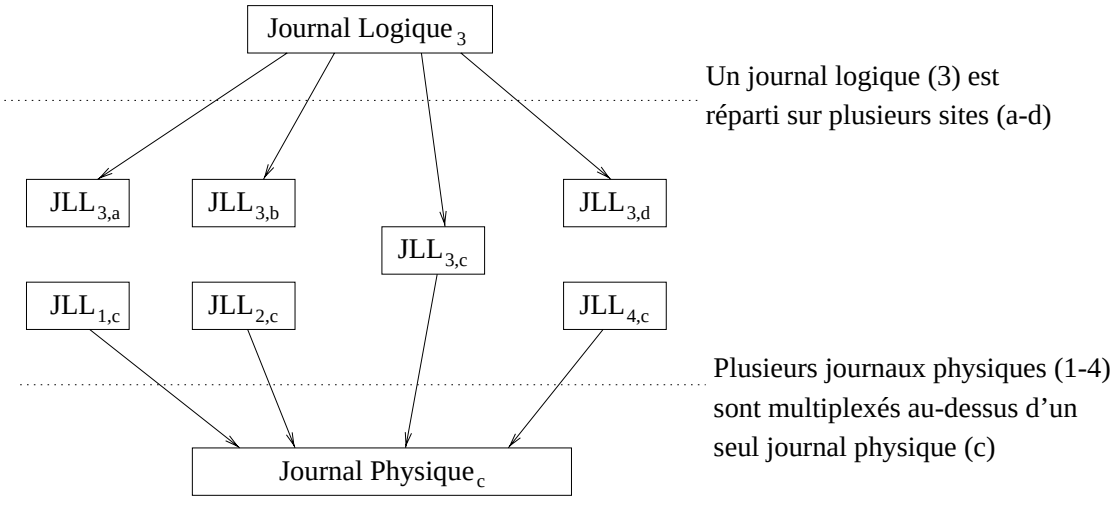

FIG. 4.2 – *Répartition de journaux*

Afin d'assurer l'aptitude des différents sites à récupérer leurs données de manière

indépendante, les informations concernant l'état de validation des journaux sont stockés dans chaque JLL qui font partie du JL. La cohérence entre les différents JLL est assurée par un *protocole de validation à deux phases* : Pendant la première phase, un *enregistrement de préparation* est écrit dans chaque JLL. Cet enregistrement de préparation ne peut être écrit que quand tous les enregistrements de données de ce journal ont été écrits. Quand tous les sites participant à la transaction ont confirmé l'écriture de leur enregistrement de préparation (et donc de tous les enregistrements de données qui le précèdent), le site coordonnateur de la transaction progresse vers la seconde phase du protocole, pendant laquelle tous les sites écrivent leur *enregistrement de validation*.

Si un JLL ne contient pas d'enregistrement de préparation après une panne, ceci signifie qu'aucun autre JLL de cette même transaction n'est validé, et donc tous les enregistrements de ce JLL peuvent être écartés.

Si au contraire, un JLL contient un enregistrement de validation, ceci signifie que tous les autres JLL de cette transaction sont au moins préparés, et contiennent donc tous les enregistrements de données. Donc la transaction peut être considérée comme validée.

Si le JLL est dans la « zone grise », c'est-à-dire s'il est préparé, mais non validé, nous devons contacter d'autres sites qui ont participé à la même transaction, jusqu'à ce que nous trouvions un site sur lequel la transaction est soit validée, soit abandonnée soit non préparée. Heureusement, cette situation est rare, et il est donc possible, dans la plupart des cas, de réparer les segments d'un site, même si les autres sites sont inaccessibles.

4.3.2 Journalisation avant

La *journalisation avant* effectue les changements sur l'image stable dès que l'application les génère. Le journal contient des enregistrements décrivant l'*état avant* le changement. Ces enregistrements permettent donc de défaire la transaction en cas d'abandon.

Si une panne arrive avant que la transaction n'ait été validée, aucune action n'est nécessaire lors de la reprise, étant donné que l'image stable est déjà en cohérence avec le journal.

Si au contraire une panne arrive avant que la transaction ne puisse être validée, tous les changements induits par cette transaction doivent être défaits.

Nous rappelons ici que la journalisation avant est aussi un cas particulier de *WAL*, tout comme la journalisation après : Les modifications sur l'image stable ne peuvent se faire qu'une fois les enregistrements correspondants du journal (ici : l'état avant des données modifiées) ont été physiquement écrits. Dans le cas de la journalisation avant, nous avons donc besoin d'écritures synchrones, autres que la préparation et la validation, qui dans ce cas se passent beaucoup plus tard. C'est pourquoi le SGJ fournit une primitive de synchronisation explicite.

Notre service de journalisation supporte donc les 2 styles de journalisation (avant et après).

4.3.3 Esquisse de l'algorithme de récupération

Notre algorithme de récupération opère en trois passes :

- Une passe de *reconnaissance*,
- Une passe de *rejeu*,
- Une passe d'*annulation*,

Pendant la passe de reconnaissance, le gestionnaire de reprise, qui fait partie du SGJ, prend l'initiative. Il examine l'état de chaque JLL, et garde cet état en mémoire. Aucune action sur l'image stable n'est effectuée à ce point. Comme l'illustre la figure 4.3, un JLL peut se trouver dans un des 3 états suivants : *abandonné, validé* ou *indéfini*. Un JLL est considéré comme abandonné, s'il contient un enregistrement d'abandon explicite, mais aussi s'il ne contient pas d'enregistrement de préparation. Un JLL est validé s'il contient un enregistrement de validation. Un JLL est dans l'état indéfini s'il contient un enregistrement de préparation, mais pas d'enregistrement de validation, ni d'abandon. À la fin de cette passe de reconnaissance, l'état des JLL est envoyé vers les autres sites participants (s'ils sont accessibles).

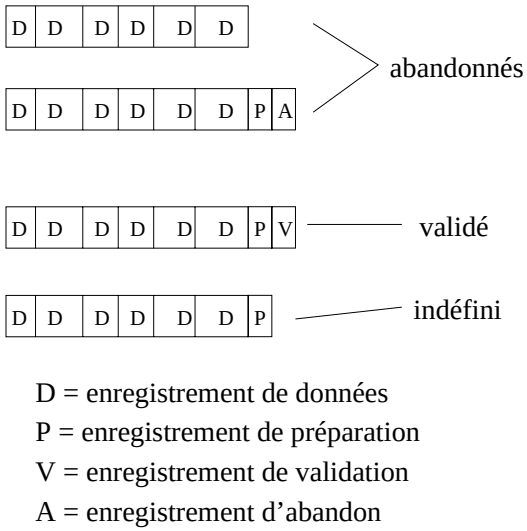

FIG. 4.3 – *États des journaux logiques après une panne*

Pendant la seconde passe, les transactions validées sont rejouées. Afin de faire ceci, le journal est parcouru dans l'ordre chronologique (du plus ancien enregistrement vers le plus nouveau). En effet, nous devons être sûr de rejouer les enregistrements les plus récents en dernier, au cas où les différents enregistrements toucheraient les mêmes données.

Pendant la troisième passe, les transactions abandonnées sont défaites. Pour des raisons similaires, le journal est parcouru en ordre chronologique inverse.

Il se peut qu'il y ait des dépendances entre ces dernières deux passes, si un même segment a été modifié à la fois par des transactions utilisant un *journal avant* et par d'autres transactions utilisant un *journal après*. À notre connaissance, ce problème n'a pas de solution propre, nous imposons donc la restriction que chaque segment

soit uniquement touché par un seul type de transaction. Si une application viole cette spécification, les résultats sont indéfinis. Cette restriction permet au système de faire tourner ces deux passes en parallèle, en utilisant deux curseurs se déplaçant dans des sens opposés.

Le SGJ est uniquement responsable de décider quels enregistrements seront défaits, rejoués ou ignorés. L'action de défaire ou rejouer elle-même est faite par les PJS. Afin de mettre en œuvre ce partage de travail, le SGJ fait des appels ascendants, pour chaque enregistrement, vers le PJS qui en est responsable.

4.3.4 Souplesse et modularité

Les journaux sont récupérés enregistrement par enregistrement. La récupération est faite par des modules qui font partie du PJS. Nous rappelons ici que le PJS est fourni ou choisi par l'application. Le SGJ, fourni par le système, gère uniquement la coordination entre sites, et prend la décision de savoir si un journal donné est validé ou pas. Afin de rendre possible ce partage de travail, chaque enregistrement est composé de deux parties : un en-tête et une partie « données ». L'en-tête est standardisé, et il peut être interprété et par le gestionnaire de récupération, et par tous les PJS. La partie données est spécifique au PJS. Il est donc possible de décrire les modifications d'une manière qui arrange l'application le mieux. Dans des cas simples, ces données peuvent être simplement une image de la zone modifiée (avant ou après), dans d'autres cas, elles peuvent être une description de plus haut niveau, tel que « déclarons cet objet comme actif ». Cependant, afin d'entrer dans notre modèle, la partie données doit remplir deux conditions :

- Comme la récupération est faite enregistrement par enregistrement, chaque enregistrement doit être autonome (*self-contained*),
- Afin de nous protéger contre des pannes qui pourraient se produire pendant la récupération, les modifications décrites par nos enregistrements doivent être *idempotentes*.

4.3.5 Gestion des enregistrements qui sont dans un état non défini

Nous voulons qu'après une panne, notre système redevienne disponible aussi rapidement que possible. Aussi ne voulons nous pas bloquer le processus de récupération entier juste à cause d'un seul JLL qui est dans un état non défini. Nous voulons bloquer uniquement la récupération de ce journal là, mais pas celle des autres.

Afin de faire ceci, nous sautons temporairement les enregistrements des JLL qui sont dans un état non défini. Dès que les autres sites redeviennent disponibles, et peuvent nous fournir suffisamment d'informations pour lever l'ambiguïté sur l'état de ces journaux mis en attente, nous pouvons reprendre leur récupération.

Cependant, nous devons être prudents d'appliquer les changements décrits par les enregistrements dans le bon ordre pour chaque donnée. D'un autre côté, l'ordre de récupération d'enregistrements relatifs à des zones de données disjointes n'est pas important. Nous résolvons le problème de l'ordonnancement de la récupération en

bloquant temporairement les segments qui ont été touchés par des journaux dans un état indéfini. Des enregistrements de journaux relatif à des segments bloqués ne sont pas repris immédiatement, même s'ils font partie d'un journal qui est dans un état défini. Dès que l'état d'un journal précédemment non défini devient défini, nous pouvons aussi récupérer les segments bloqués à cause de lui.

4.4 Interaction avec la mémoire virtuelle

Afin d'améliorer les performances de notre système, nous intégrons la gestion de la mémoire virtuelle avec la gestion du stockage. Dans cette section, nous montrons comment cette intégration nous permet de gagner de la place en mémoire d'exécution, et de faire un meilleur usage de la bande passante vers le disque.

Dans la conception initiale de Arias, des données chargées en mémoire d'exécution y restaient «éternellement», c'est-à-dire jusqu'à une panne. L'inconvénient de cette approche est que la mémoire d'exécution doit être énorme afin de pouvoir contenir toutes ces données qui s'y accumulent au cours du temps. Nous avons donc décidé que ces données pourraient être retirées de la mémoire d'exécution, sous certaines conditions.

4.4.1 Mise en œuvre simple, qui utilise des pages entières

Dans cette approche, nous éliminons uniquement des pages «propres». Une page est propre si elle contient uniquement des zones propres. Des zones propres sont des zones qui n'ont jamais été modifiées. Des zones qui ont déjà été écrites sur le stockage stable, et qui n'ont pas été modifiées depuis sont considérées comme propres elles aussi. Nous avons utilisé une méthode similaire dans notre système précédent (Guide [Che92]).

4.4.2 Mises à jour de l'image en différé

Quand des applications (ou leur PJS) ont écrit leurs mises à jour dans le journal, et validé le journal, elles n'ont pas besoin de répercuter ces modifications sur l'image immédiatement. En effet, dans le cas d'une panne, nous serons capables de rejouer ces opérations à partir du journal. Du moment que ces opérations ont été écrites sur le journal, nous pouvons permettre au système de choisir lui-même l'instant où il va écrire ces données sur l'image. Ceci peut être tout de suite après la validation du journal, ou à un instant ultérieur, si actuellement la charge sur le sous-système de stockage est trop élevée.

4.4.3 Définition de l'état cohérent

Nous définissons un troisième état, en plus des états sale et propre. C'est l'état *cohérent*. Une zone est considérée comme cohérente si la dernière modification de cette zone a déjà été consignée sur le journal. Une zone cohérente peut être écrite sur l'image sans risquer une incohérence. En effet, par définition, une zone cohérente est en accord avec le journal, et peut donc être récupérée après une panne, qu'elle

ait été répercutée sur l'image ou non. Si nécessaire, la recopie peut avoir lieu après la panne.

Le système fournit un point d'entrée qui permet au PJS de déclarer une zone comme cohérente. D'habitude, ce point d'entrée est invoqué après la validation du journal qui décrit la modification apportée à la zone en question.

4.4.4 Granularité

La granularité de la gestion de mémoire matérielle est la page. Cependant, dans Arias, nous utilisons un grain plus fin : la zone, qui contient un seul objet. Une page peut donc contenir un mélange de zones propres et sales. Nous devons suivre l'état de chaque zone de la page. Une page peut seulement être vidée de la mémoire d'exécution si toutes ses zones sont propres.

4.4.5 Réutilisation d'une zone cohérente

Si une zone cohérente est à nouveau salie par une transaction suivante, le système a le choix entre plusieurs manières de procéder :

- Il peut forcer une écriture immédiate de cette zone sur l'image, avant d'accorder le verrou. Malheureusement, ceci augmente la latence de verrouillage.

- Il peut simplement annuler l'écriture prévue en permettant à la zone de passer directement de l'état cohérent à l'état sale. Visiblement, ceci soulève la question du relancement de cette écriture plus tard. En effet, l'application qui a fait la seconde modification (celle qui a sali la zone) ne la redéclare pas nécessairement comme cohérente dans un délai raisonnable. De plus, une zone très sollicitée peut être réquisitionnée encore et encore par de nouvelles applications, avant qu'elle ne puisse être écrite sur l'image stable, et ceci même si toutes ces applications déclarent la zone comme cohérente quand elles ont fini. Ceci implique que les enregistrements de journaux qui décrivent des modifications sur cette zone ne deviennent jamais obsolètes, et continuent donc à jamais de consommer de l'espace disque. On pourrait pallier à ce problème en faisant expirer les enregistrements du journal qui décrivent des modifications, par une transaction précédente, de ces zones éternellement sales dès que la transaction courante valide (et écrase donc la modification de la transaction précédente). Cependant, ceci ne résoudrait pas le problème, étant donné que les différentes transactions pourraient utiliser des zones qui se recoupent, mais ne sont pas rigoureusement identiques. De plus, comme notre journal a un format souple, la description de la dernière modification seule n'est donc pas nécessairement suffisante pour en déduire l'état final de la zone.

- Le système pourrait aussi faire une copie en mémoire de la zone, et démarrer une écriture immédiate de la zone à partir de cette copie. Le verrou pourrait être accordé dès que cette copie est faite (latence faible), sans attendre la complétion de l'écriture sur disque. Cependant, les enregistrements en question peuvent seulement être déclarés comme obsolètes une fois que l'écriture est effectivement terminée. Nous adoptons cette dernière solution.

4.4.6 Algorithme de vidage de pages

Quand le paginateur a besoin de libérer de l'espace de mémoire, il choisit une page à vider, et il y applique les opérations suivantes :

1. Il regarde d'abord si la page contient des zones cohérentes. S'il y en a, il les écrit sur l'image stable, et les fait ainsi passer à l'état propre,
2. Puis il regarde si toutes les zones de la page sont maintenant propres :
   - Si oui, la page peut être éliminée de la mémoire centrale sans que l'on ait besoin de la sauvegarder dans l'espace de swap.
   - Si au contraire la page contient des zones sales, elle doit être écrite dans l'espace de swap.

Les pages à vider sont choisies selon l'ordre *LRU (Least recently used)*. Afin d'améliorer les diverses performances, on pourrait biaiser cet algorithme en faveur des pages propres, ou en faveur des pages contenant beaucoup de zones cohérentes, ou bien en faveur des pages contenant des zones cohérentes particulièrement vieilles. Il y a de bons arguments pour chacun de ces choix :

- Vider des pages propres n'engendre pas d'entrées-sorties supplémentaires,
- Vider des pages contenant beaucoup de zones cohérentes permet d'expirer un nombre maximal d'enregistrements.

Des mesures de performances ultérieures nous permettrons de trancher entre ces différents choix.

4.4.7 Diagramme de transition d'états

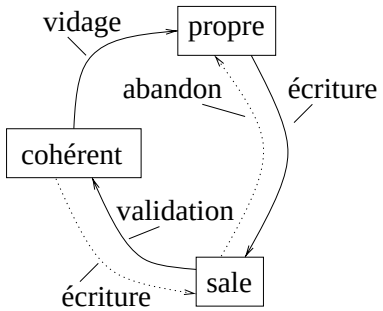

FIG. 4.4 – *Diagramme de transitions d'états d'une page*

La figure 4.4 montre les transitions suivantes :

- Une page propre devient sale quand une transaction la modifie,
- Une page sale devient cohérente quand la transaction qui l'a salie valide,
- Une page cohérente devient propre quand elle est écrite sur l'image stable,

- Une page cohérente peut redevenir sale, si une seconde transaction y écrit (nous ne faisons pas ceci : comme illustré dans la sous-section 4.4.5, nous l'écrivons d'abord sur disque dans ce cas là),
- Une page sale peut redevenir propre si la transaction qui l'a salie abandonne volontairement. Dans ce cas, soit les données originales sont rechargées à partir de l'image stable afin de remettre la page dans l'état (journalisation après), soit les modifications sont défaites grâce au journal avant.

4.5 Purge d'enregistrements obsolètes

Dans cette section, nous décrivons comment nous vidons des enregistrements obsolètes du journal. Un enregistrement est considéré comme obsolète s'il décrit un changement qui a déjà été répercuté sur l'image stable.

4.5.1 Contexte général

En théorie, le journal physique peut être considéré comme infini. Cependant, en réalité, le journal a seulement une taille finie, et nous devons être capables de réutiliser l'espace occupé par des enregistrements qui ne sont plus utiles. Ces enregistrements inutiles sont appelées *obsolètes*. Si nous désirons avoir la capacité d'annuler des transactions après leur validation, les enregistrements obsolètes peuvent être archivés hors-ligne sur du stockage tertiaire (bandes magnétiques). Si nous n'avons pas besoin de cette faculté, les enregistrements obsolètes peuvent tout simplement être éliminés du journal. L'action de déclarer des enregistrements comme obsolètes est appelée la *purge* du journal. Il y a deux grains possibles pour la purge :

- Nous faisons expirer seulement des JLL entiers,
- Nous faisons expirer des enregistrements individuels.

Nous choisissons le dernier niveau de granularité, comme il nous donne plus de souplesse. Ce regain de souplesse est d'autant plus désirable que nous avons renoncé de compacter le journal pour des raisons de performance.

Comme le montre la figure 4.5, nous pouvons libérer la partie la plus ancienne du journal qui soit entièrement constituée d'enregistrements obsolètes. Nous espérons que les enregistrements deviennent obsolètes dans à peu près le même ordre que celui dans lequel ils ont été écrits. Heureusement, en général cette hypothèse est vraie.

4.5.2 Le déroulement de la purge du journal en fonctionnement normal

4.5.2.1 Le rôle du PJS

Lorsque le PJS envoie un enregistrement au SGJ, il lui associe un identificateur, qu'il envoie ensemble avec l'enregistrement. Il garde aussi une copie de cet identificateur pour lui-même, qu'il communiquera au paginateur lorsqu'il déclarera la zone modifiée par cet enregistrement comme cohérente.

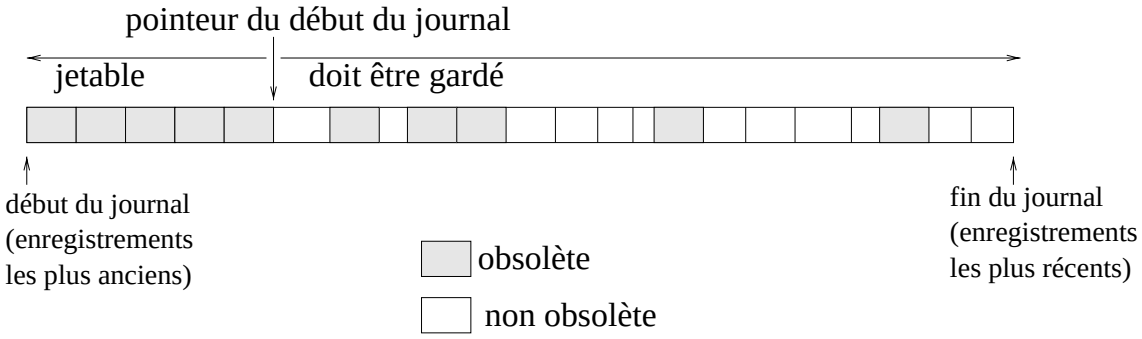

FIG. 4.5 – *Purge dy journal et début du journal*

4.5.2.2 Le rôle du paginateur

Le paginateur garde une table de correspondance qui donne, pour chaque zone cohérente, l'identificateur de l'enregistrement de journal qui décrit la dernière modification apportée à cette zone. Lorsque le paginateur a sauvegardé la zone en question sur l'image stable, il en communique l'identificateur au SGJ.

4.5.2.3 Le rôle du SGJ

Le SGJ garde en mémoire une table de correspondance qui permet de déterminer à partir de l'identificateur d'un enregistrement sa position dans le journal physique. Cette table est organisée en file d'attente, triée selon les positions croissantes[16], afin de pouvoir facilement déterminer quel est le plus vieil enregistrement qui n'est pas encore répercuté sur l'image. Cet enregistrement marque la limite entre les enregistrements obsolètes et les enregistrements non obsolètes.

4.5.2.4 Nature des identificateurs

Les identificateurs sont obtenus par concaténation de l'identificateur du JL (que le PJS obtient lors de la création du JL), d'un numéro de séquence local à ce journal et éventuellement du numéro du site client émetteur. Ceci permet au PJS de générer des identificateurs qui sont globalement uniques.

4.6 Disponibilité et durabilité

4.6.1 Réplication

L'image stable assure la durabilité des données, à condition qu'il n'y ait pas de pannes de disque. La *réplication de données* sur plusieurs disques permet de se protéger contre ce genre de pannes : Si un exemplaire des donnés est en panne, on accède l'autre. De plus, la réplication permet d'obtenir une meilleure disponibilité, aussi en cas de panne temporaire de la machine à laquelle le disque est attaché. La réplication permet de garder le système distribué opérationnel et disponible en

16. Cette structure sera décrite en détail dans la section 5.6.

dépit de pannes partielles. Cependant, afin de garantir une sémantique correcte en cas de partition de réseau, des protocoles de synchronisation complexes et souvent coûteux sont nécessaires afin de garder les répliques en phase. En général, le coût des opérations dans des bases de données répliquées augmente, étant donné que le système doit éventuellement accéder à des copies multiples d'un même objet afin de garantir la cohérence mutuelle des copies.

Le protocole de réplication le plus simple est le protocole *read-one write-all*. Dans ce protocole, une lecture est réalisée en accédant n'importe laquelle des répliques (en général la plus proche). Une écriture nécessite une mise à jour de toutes les copies, afin d'assurer que la prochaine lecture retourne la valeur appropriée de l'objet. L'inconvénient de cette approche est que la disponibilité n'est pas vraiment améliorée pour les écritures : en effet, la panne d'une seule des répliques interdit toute évolution des données répliquées. Par contre, les accès en lecture seule restent possible tant qu'il y a au moins un exemplaire accessible.

Afin d'améliorer la disponibilité, des protocoles basés sur le *vote* ont été proposés. Dans le protocole de vote statique [Tho79], une opération d'écriture écrit Q_e copies, et une opération de lecture accède à Q_l copies. L'écriture réussit seulement si au moins Q_e copies sont accessibles, et la lecture réussit seulement si au moins Q_l copies sont accessibles et identiques entre elles.

Q_e et Q_l sont appelés *quorum d'écriture* et *quorum de lecture* respectivement, et sont choisis de manière à ce que $Q_e + Q_l$ soit supérieur au nombre total des exemplaires de l'objet. Cette contrainte protège l'algorithme contre les partitions de réseau. En effet, à cause de cette relation numérique, il y a parmi les Q_l copies lues au moins une des Q_e copies écrites précédemment. Donc si la lecture est possible (accord entre copies lues), son résultat est obligatoirement la dernière valeur écrite. Afin de se protéger contre des mises à jour contradictoires et cohérentes, deux approches sont possibles :

- Le système peut exiger qu'une lecture soit fait immédiatement avant l'écriture,
- Le système peut imposer la contrainte supplémentaire que $Q_e > \frac{n}{2}$.

Ce système diminue la disponibilité pour la lecture au prix de la disponibilité pour l'écriture. Il existe des variantes de ce protocole, qui introduisent un des changements suivants :

- On peut associer des poids au sites [Gif79]. Ainsi un site qui est connu comme étant hautement disponible peut par exemple compter pour 3 dans le calcul du quorum,
- On peut introduire des *votants phantômes* qui ne stockent pas de copie sur disque, mais qui servent uniquement à augmenter la disponibilité grâce au vote qu'ils peuvent émettre. Évidemment, les quorums devront être choisis de telle manière qu'une majorité peut être atteinte sans les votants phantômes.

Dans Arias, nous n'avons pour l'instant pas mis en œuvre de protocole de réplication, mais nous prévoyons éventuellement de faire ceci plus tard.

5

Mise en œuvre du service de stokage

Dans ce chapitre, nous décrivons la structuration interne du service de journalisation et de stockage.

5.1 Architecture interne du sous-système de stockage

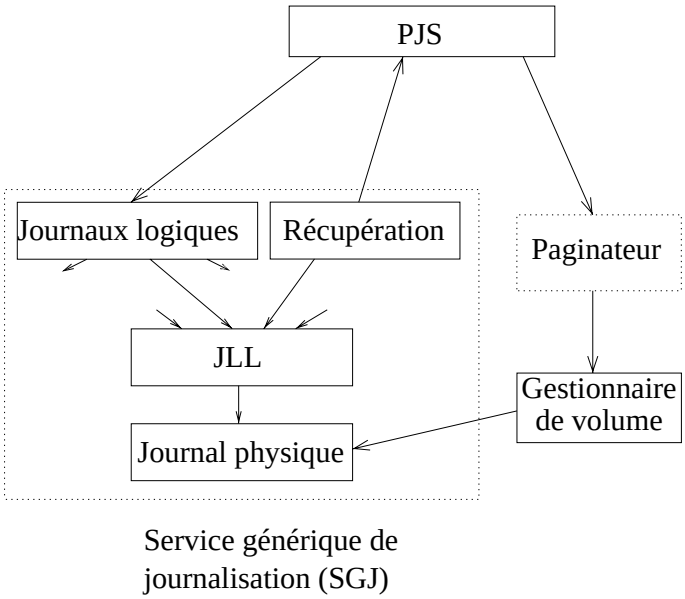

FIG. 5.1 – *Architecture interne du service de stockage*

Comme le montre la figure 5.1, le service de stockage met en jeu les modules suivants :

- Le gestionnaire de volume est responsable du stockage de l'image stable des données.

- Le PJS prend, pour le compte de l'application, l'initiative des opérations de sauvegarde, et supervise tout en tant que coordonnateur.

- Le SGJ, qui est subdivisé en plusieurs parties :
  - Le *gestionnaire du journal physique (GJP)* est responsable du stockage physique des journaux. Lors de la reprise, il localise le début et la fin du journal,
  - Le *gestionnaire des journaux logiques locaux (SJLL)* est responsable de multiplexer plusieurs journaux logiques locaux au-dessus d'un même journal physique,
  - Le *gestionnaire des journaux logiques (SJL)* est responsable d'éclater un journal logique en plusieurs journaux logiques locaux, qui seront chacun stockés sur un site différent, auprès de leur images stables,
  - Le gestionnaire de la récupération est responsable de remettre en état l'image stable après une panne, en rejouant ou en défaisant les opérations décrites par le journal.

Dans la suite de ce chapitre, nous passerons en revue le rôle et la mise en œuvre de ces différents modules.

5.2 Le gestionnaire de volume

5.2.1 Atomicité des mises à jour

Bien que l'atomicité des transactions soit déjà assurée par le service de journalisation, nous devons quand même faire attention de ne pas perdre des anciennes données quand nous mettons à jour l'image stable.

L'unité minimale d'écriture sur disque, permise par le matériel, est le *secteur*. Quand on demande au système d'exploitation d'écrire des parties d'un secteur, le système lit d'abord le secteur tout entier, puis il y insère les nouvelles données dans l'image mémoire de ce secteur, et finalement il réécrit le nouvel état du secteur sur disque. Si une panne intervient lors de la réécriture du secteur sur disque, le secteur complet devient illisible, et on perd toutes les données qui y ont été stockées, y compris celles qui n'ont pas été mises à jour. Évidemment, il n'y a pas d'enregistrement de journal pour les données non touchées. Si on ne prend pas des dispositions appropriées, ces données sont donc perdues à jamais dans un tel scénario.

Afin de tenir compte de ce genre de problème, le gestionnaire de volume ne doit pas réécrire les nouvelles données au même endroit que les anciennes. Une manière d'éviter ceci est de garder un mini-journal géré au sein du gestionnaire de volume, qui stocke les *images après* des secteurs. Ce journal peut être de taille très réduite, car le gestionnaire de volume n'a pas besoin d'attendre la validation d'une transaction avant de pouvoir expirer ses enregistrements. C'est cette solution qui a été adopté dans le JFS de AIX que nous utilisons comme support pour le stockage des volumes.

5.3 Le journal physique

5.3.1 Étendue du journal

Afin d'en simplifier la gestion, notre journal physique est stocké de manière circulaire sur disque, de telle manière que si le curseur d'écriture atteint la fin physique de la partition, il recommence au début de cette partition. L'étendue du journal est donc décrit par deux curseurs, celui du début du journal (qui correspond aux enregistrements les plus anciens), et celui de la fin (qui correspond aux enregistrements les plus récents. Le début bouge quand des enregistrements deviennent obsolètes, et la fin bouge quand de nouveaux enregistrements sont écrits.

Nous n'utilisons pas de compactage coûteux, car nous avons constaté que les enregistrements deviennent obsolètes à peu près dans le même ordre que celui dans lequel il sont écrits.

5.3.2 Localisation de la fin du journal

5.3.2.1 Les problèmes et les solutions simplifiées

Le gestionnaire du journal physique doit stocker le journal de manière à faciliter la localisation des deux bornes lors de la reprise. Ceci doit être fait de façon à assurer une bonne bande passante, et une latence aussi faible que possible pour les opérations synchrones (validation, préparation et synchronisation).

La fin du journal doit être trouvée de manière exacte. En effet, les opérations de synchronisation garantissent, une fois qu'elles rendent la main au PJS, que le données ont été écrites physiquement sur le support de stockage, et restent visibles après une panne. Cette propriété est indispensable pour assurer la permanence des transactions validées. Par conséquent, la fin du journal, telle qu'elle est vue après une panne, doit être située au-delà des dernières écritures déclenchées par des opérations synchrones.

Une solution simple serait d'écrire la position des limites du journal après chaque écriture à un endroit fixe. Un tel endroit fixe est appelé *ancre*. L'inconvénient de cette approche est que pour chaque écriture logique d'un enregistrement, il faut deux écritures physiques : celle de l'enregistrement lui-même, et celle de l'ancre. Ceci est d'autant plus défavorable que les deux secteurs sont éloignés les uns des autres, donnant lieu ainsi à d'amples mouvements du bras du disque. Cette solution simpliste n'est donc pas acceptable pour nos besoins.

Une autre solution un peu plus évoluée serait d'adopter la convention que des blocs non utilisés sont remplis de zéros. L'inconvénient de cette approche est d'abord qu'il y a ambiguïté, car une application peut très bien décider d'écrire un enregistrement plus grand qu'un secteur, et qui est entièrement rempli de zéros. De plus, lors de la purge du journal, il faudrait remettre les données purgées à zéro. Ceci engendre aussi deux écritures physiques par écriture logique : en effet, le coût occasionné par la purge est proportionnel au nombre de secteurs écrits (chaque secteur sera purgé un jour), et doit donc aussi être comptabilisé dans le coût total de l'écriture.

5.3.2.2 Notre solution

La solution que nous adoptons est basée sur des numéros de génération stockés dans chaque secteur. Chaque secteur porte à sa fin un entier qui compte le nombre de parcours entiers qui ont été faits depuis la création de ce journal physique.

Lors de l'opération normale, nous écrivons périodiquement la position de la fin du journal, ainsi que le numéro de génération courant (dit *de référence*) dans l'ancre. Cette écriture est faite de manière relativement rare, afin de limiter les mouvements excessifs du bras de disque. La fin du journal, telle qu'elle est enregistrée dans l'ancre, n'est donc qu'une indication approximative. La seule contrainte qu'elle doit vérifier est qu'elle fasse partie du journal, c'est-à-dire qu'elle ne se fasse pas «dépasser» par le début du journal, qui évolue au cours des purges.

La figure 5.3 montre l'algorithme de localisation de la fin du journal physique. Cet algorithme utilise les deux variables `FinApproximative` et `GénérationRéférence` qui sont lues à partir de l'ancre avant l'algorithme proprement dit.

La figure 5.2 illustre l'état des numéros de génération dans chaque secteur du journal dans deux cas différents. La partie gauche montre un journal physique qui «ne fait pas le tour» de la partition physique, alors que celle de droite montre un journal «qui fait le tour».

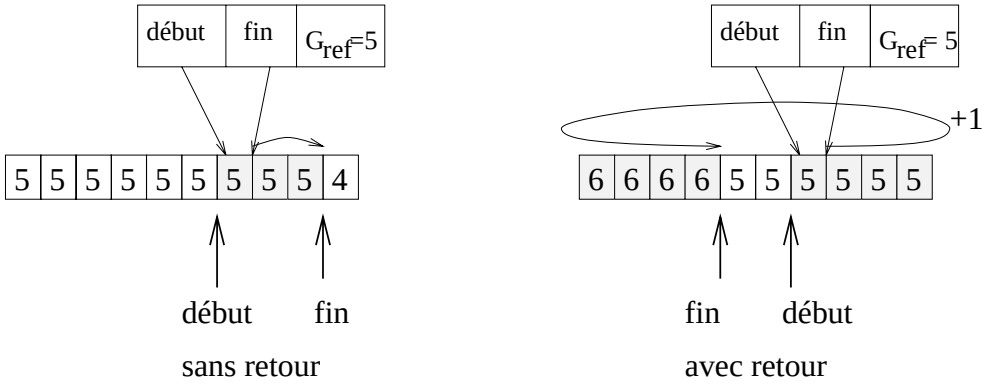

FIG. 5.2 – *Localisation de la fin du journal avec et sans retour*

5.3.3 Le début du journal

Notre algorithme de localisation du début est basé sur le principe que nous n'avons pas besoin de localiser le début du journal de manière aussi précise que sa fin. Si le début pointe vers des enregistrements trop vieux, le seul effet négatif serait que nous rejouions, après une éventuelle panne, des enregistrements qui ont déjà été répercutés. Ceci n'entraîne pas d'incohérences étant donné que nos enregistrements sont idempotents. Le seul effet négatif est une légère perte de performance de reprise, due au fait que nous faisons des opérations qui ont déjà été faites. Cette perte de performance à la reprise ne nous gêne pas, car nous considérons les pannes comme des événements rares et nous optimisons le cas commun (journalisation) au dépens du cas exceptionnel (panne).

D'un autre côté, nous devons cependant faire attention de ne jamais faire pointer le début vers des enregistrements trop jeunes. En effet, une erreur dans ce sens-là

```
Fonction TrouveFin1 : TypeCurseur ;
   CursFin ← FinApproximative ;
   GénCourante ← GénRéférence ;
   tant que GénCourante = Génération(CursFin) faire
      CursFin ← ProchainSecteur(CursFin) ;
      si CursFin ← FinPhysique alors ;
         /* Nous augmentons le numéro de génération courante
          * pour refléter ce qui a été fait lors de l'écriture */
         GénCourante ← GénCourante + 1 ;
         CursFin ← DébutPartition ;
      fin si
   fin tant que
```

FIG. 5.3 – *Algorithme de localisation de la fin de la partition physique*

entraînerait la perte de ces enregistrements expirés de manière prématurée après une panne.

Nous mettons donc à jour très rarement le début du journal tel qu'il est stocké dans l'ancre. Une mise à jour en différé peut seulement faire pointer le pointeur du début vers des enregistrements trop vieux, mais jamais vers des enregistrements trop jeunes, donc d'après les paragraphes précédents, cette approche garde la cohérence du journal, et permet d'obtenir des performances en journalisation qui sont acceptables.

5.3.4 Sécurité de l'ancre du journal

Périodiquement, les pointeurs de fin et de début du journal, ainsi que le numéro de génération courant sont écrits dans un endroit fixe du journal, appelé ancre. Afin de nous protéger contre les pannes qui pourraient survenir pendant l'écriture de l'ancre, et qui rendraient l'ancre illisible (cf. section 5.2.1), nous stockons l'ancre en deux exemplaires. Nous mettons à jour alternativement l'un et l'autre. Chacun des deux exemplaires porte une estampille, incrémentée lors de chaque mise à jour. Comme l'illustre la figure 5.4, lors de la reprise après panne, nous utilisons l'exemplaire non endommagé le plus récent. La variable `ProchainAncre` indique la position sur laquelle sera écrite l'ancre la prochaine fois. Elle doit notamment être positionnée de manière à éviter d'écraser l'unique exemplaire valable.

5.3.5 Sécurité des écritures et bande passante

Tout comme pour le volume et pour l'ancre, le problème des pannes pendant l'écriture physique d'un secteur se pose aussi pour le journal. La modification d'un secteur déjà écrit d'un journal peut donc, ici aussi, entraîner la perte des données déjà écrites.

Dans un contexte transactionnel, ceci est inacceptable. Nous devons donc faire en sorte d'écrire le journal **uniquement par secteurs entiers**.

```
Fonction ChoisitAncre ;
   si EstBon?(Ancre1) et EstBon?(Ancre2) alors
      si Date(Ancre1) > Date(Ancre2) alors
         Lire(Ancre1) ;
         ProchainAncre ← 2 ;
      sinon
         Lire(Ancre2) ;
         ProchainAncre ← 1 ;
      fin si
   sinon-si EstBon?(Ancre1) alors
      Lire(Ancre1) ;
      ProchainAncre ← 2 ;
   sinon-si EstBon?(Ancre2) alors
      Lire(Ancre2) ;
      ProchainAncre ← 1 ;
   sinon
      erreur(''Les deux ancres sont mauvais'') ;
   fin si
```

FIG. 5.4 – *Algorithme de choix du bon ancre*

Une méthode pour résoudre le problème est de *remplir* par des zéros les enregistrements qui seraient plus petits qu'un secteur. Malheureusement, cette méthode mène à un faible débit si les enregistrements sont nettement plus petits qu'un secteur disque[17]. Mettre un seul enregistrement de 30 octets dans tout un secteur serait donc défavorable pour la bande passante de notre système de journalisation.

Nous essayons donc de regrouper plusieurs enregistrements dans un même secteur. Pour faire ceci, nous gardons les enregistrements dans un tampon de mémoire de la taille d'un secteur. Nous écrivons ce tampon sur disque quand il est plein, ou quand nous recevons un enregistrement correspondant à une écriture synchrone. La figure 5.5 illustre l'écriture de données. `CurseurÉcriture`, `IndCassure` et `FinDernEnreg` sont des variables globales[18]. Les 3 dernières de ces variables sont utilisées par les routines de plus haut niveau pour écrire les métadonnées. Ces métadonnées seront décrites dans les sous-sections 5.3.7, 5.3.8.4 et 5.3.2.2.

5.3.6 Organisation du journal physique

Comme l'indique la figure 5.6, le journal physique contient deux flots, qui peuvent se comprendre comme deux journaux entrelacés l'un et l'autre :

1. Les données des enregistrements qui sont stockés dans les 4076 premiers octets du secteur.

17. Sur AIX, un secteur disque contient 4096 octets, alors que certains enregistrements ne contiennent que quelques dizaines d'octets.

18. Ceci est une vision simplificatrice. En réalité ce sont des membres de la structure représentant le journal physique.

```
Procédure ÉcritureDonnées(Taille, Données) ;
   tant que Taille > 0 faire
      si Taille > TailleDisponibleDansTampon alors
         TailleLocale ← TailleDisponibleDansTampon ;
      sinon
         TailleLocale ← Taille ;
      fin si
      AjouterÀTampon(Données, TailleLocale) ;
      Taille ← Taille - TailleLocale ;
      Données ← Données + TailleLocale ;
      si TamponPlein? alors
         ÉcrireTampon(Tampon, GénCourante,
                      IndCassure, FinDernEnreg) ;
         MarqueVide(Tampon) ;
         si RetourAuDébut? alors
            GénCourante ← GénCourante + 1 ;
         fin si
      IndCassure ← faux ;
      fin si
   fin tant que
```

FIG. 5.5 – *Algorithme d'écriture de données*

2. Des métadonnées, utilisées lors de la reprise. Ces métadonnées sont stockées dans les 16 derniers octets du segment, et comprennent :

   – le numéro de génération, qui a déjà été mentionné,

   – l'indicateur de cassure, qui sera décrit dans la suite,

   – un pointeur vers la fin du dernier enregistrement complet lors de l'écriture de ce secteur, qui permet de facilement repérer la fin du journal, à l'octet près, lors de la récupération. En présence d'enregistrements suffisamment grands, ce pointeur peut éventuellement pointer dans un secteur précédent.

D'autres métadonnées se trouvent dans le flot principal du journal. Elles établissent un chaînage qui lie chaque enregistrement à son prédécesseur et à son successeur. Le chaînage arrière est un pointeur sur le début de l'enregistrement précédent, alors que le chaînage avant est représenté comme taille de l'enregistrement suivant.

5.3.7 L'indicateur de cassure

L'indicateur de cassure indique si le dernier enregistrement du secteur précédent est invalide. Une cassure (*break*) peut apparaître pour l'une des deux raisons

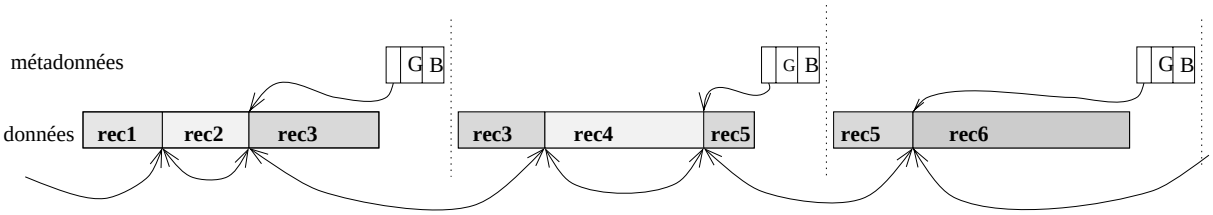

FIG. 5.6 – *Métadonnées du journal physique*

suivantes :

- Lors d'une panne qui survient pendant l'écriture d'un enregistrement multi-secteurs,

- Lors de certaines opérations de synchronisation,

5.3.7.1 Cassures lors d'une panne

Notre système n'exige pas que les enregistrements logiques soient alignés sur les secteurs physiques : nous permettons que des enregistrements se trouvent à cheval entre deux secteurs successifs. La seconde partie de l'enregistrement est transférée de la mémoire tampon sur le disque quand le secteur dans lequel elle se trouve est entièrement rempli. Si une panne arrive dans cette situation, nous allons retrouver, lors de la reprise, un enregistrement à moitié écrit. Le pointeur de fin de dernier enregistrement, qui fait partie des métadonnées, nous permet de localiser la fin logique du journal sans problème. Mais, comme nous avons vu dans la sous-section 5.3.5, nous ne pouvons plus ajouter des données dans le même secteur. Les nouveaux enregistrements, qui seront générés après la panne doivent donc être écrits dans le secteur suivant. Cet état de choses doit être marqué dans le journal, afin que la récupération d'une éventuelle seconde panne puisse se retrouver. C'est à quoi sert l'indicateur de cassure. Cet indicateur est positionné à 1 dans le premier enregistrement écrit après la panne, si le journal tel qu'il a été retrouvé après la panne se termine par un enregistrement partiel.

5.3.7.2 Cassures lors des synchronisations

Quand le PJS nous demande de faire une opération synchrone (validation, préparation, abandon ou synchronisation), nous devons attendre que l'enregistrement soit physiquement écrit sur disque avant que nous puissions confirmer l'opération à l'application. Or l'écriture physique ne peut se faire qu'une fois que le secteur tampon est rempli entièrement. Comme décrit dans la sous-section 5.3.5, nous essayons de remplir le secteur par des enregistrements, synchrones ou asynchrones d'autres transactions. Mais, parfois il arrive qu'il n'y a pas d'autre transaction. Dans ce cas, nous remplissons le secteur par des zéros, afin de garantir une écriture dans un délai raisonnable, pour de ne pas trop faire attendre le PJS. Ces zéros sont sauvegardés sous forme d'un enregistrement de taille supérieur à un secteur disque, qui sera marqué comme invalide par l'indicateur de cassure du secteur suivant.

Après une cassure les enregistrements logiques reprennent à partir de l'octet 16 : les 8 premiers octets sont occupés par un lien arrière, tandis que les 8 suivants constituent la taille du nouvel enregistrement.

5.3.8 Algorithmes

Dans les algorithmes qui sont décrits dans la suite, nous supposons faites les procédures de lecture et d'écriture de bas niveau. Les procédures `LireEntier` et `Lire` lisent les données à partir du flot principal des données, tandis que `Cassure?`, PointeurDernier accèdent aux métadonnées concernées. L'unité pour les curseurs est l'octet. Nous suppose aussi que les primitives utilisées masquent la nature circulaire du journal physique.

5.3.8.1 Écriture

La figure 5.7 illustre l'algorithme d'écriture. Cet algorithme présuppose l'existence de primitives d'écriture qui gèrent correctement le tampon de secteur.

```
Procédure ÉcrireEnreg(Curseur, Données, Taille) ;
   FinDernEnreg ← Curseur ;
   CursDébut ← Curseur ;
   ÉcrireEntier(Curseur,Taille) ;
   Curseur ← Curseur + 8 ;
   ÉcrireDonnées(Curseur, Données, Taille) ;
   Curseur ← Curseur + Arrondi8(Taille) ;
   /* cet arrondissement est nécessaire afin de garantir un bon
    * alignement des enregistrements suivants */
   ÉcrireEntier(Curseur, CursDébut) ; /* chaînage arrière */
   CursFin ← Curseur + 8 ;
   FinProcédure
```

FIG. 5.7 – *Algorithme d'une écriture d'un enregistrement*

5.3.8.2 Lecture en avant

Pour lire en avant à partir de la position du curseur, nous devons d'abord nous assurer que l'enregistrement sous le curseur est valide. Un enregistrement tenant dans un seul secteur est toujours valide, alors qu'un enregistrement occupant plusieurs secteurs est seulement valide s'il n'y a pas de cassure.

Pour lire un enregistrement vers l'avant, nous déroulons l'algorithme illustré dans la figure 5.8. La boucle extérieure est nécessaire, car en cas de pannes très peu espacées, il peu arriver que le nouvel enregistrement commençant au-delà de la cassure soit invalide lui aussi. La fonction retourne un curseur qui pointe vers l'enregistrement suivant. Ce curseur est arrondi vers le multiple de 8 le plus proche, car un tel arrondi a aussi été fait lors de l'écriture.

```
Fonction LireSuiv(CursDébut : TypeCurseur) : TypeCurseur ;
   Taille ← LireEntier(CursDébut) ;
   si MêmeSecteur?(CursDébut, CursDébut + Taille + 16) alors
      /* l'enregistrement se termine dans le même secteur ; il est
       * donc forcément valide */
      Lire(CursDébut + 8, Taille) ;
      retour Arrondi8(CursDébut+Taille+16) ;
   sinon
      DébutBoucle BoucleExtérieure
         CursFin ← DébutProchainSecteur(CursDébut) ;
         DébutBoucle BoucleIntérieure
            si CursFin > CursDébut + Taille + 8 alors
               /* L'enregistrement a été lu entièrement sans
                * cassure */
               Lire(CursDébut + 8, Taille) ;
               retour Arrondi8(CursDébut+Taille+16) ;
            fin si
            si ¬ DansLeJournal?(CursFin) alors
               erreur(''Fin du journal atteinte'') ;
            fin si
            si Cassure?(CursFin) alors
               /* Enregistrement invalide, reprendre après
                * cassure */
               CursDébut ← CursFin + 8 ;
               Taille ← LireEntier(CursDébut) ;
               ProchaineItération BoucleExtérieure ;
            fin si
         FinBoucle BoucleIntérieure ;
      FinBoucle BoucleExtérieure ;
   fin si
```

FIG. 5.8 – *Lecture de l'enregistrement suivant, en tenant compte des cassures*

5.3.8.3 Lecture en arrière

La figure 5.9 représente l'algorithme de lecture en arrière. Nous n'avons pas besoin de nous préoccuper de cassures dans ce cas. En effet, le pointeur de chaînage arrière est toujours exact, car à l'instant où il est écrit, nous savons déjà s'il y a eu une panne ou pas. Cependant, la taille de l'enregistrement, qui figure avant l'enregistrement, doit être lue aussi, car elle ne peut pas être déduite directement du pointeur de chaînage arrière pour les deux raisons suivantes :

- La taille peut ne pas être un multiple de 8 et il n'y a aucune trace des octets de remplissage dans le pointeur de chaînage.

- Il peut y avoir plusieurs enregistrements cassés qui suivent l'enregistrement lu.

```
Fonction LirePrec(CursFin : TypeCurseur) : TypeCurseur ;
   CursFin ← CursFin - 8 ;
   CursDébut ← LireEntier(CursFin) ;
   si ¬ DansLeJournal?(CursFin) alors
      erreur(‘‘Début du journal atteinte’’) ;
   fin si
   Taille ← LireEntier(CursDébut) ;
   Lire(CursDébut + 8, Taille) ;
   retour CursDébut ;
```

FIG. 5.9 – *Lecture de l'enregistrement précédent, en tenant compte des cassures*

5.3.8.4 Reprise après panne

Lors d'une reprise après panne, nous devons d'abord localiser l'adresse du dernier enregistrement valable, afin de l'utiliser comme point de départ pour la première passe décrite dans la sous-section 4.3.3.

Mais nous devons aussi mettre le journal dans un état tel qu'il puisse accepter de nouvelles écritures. Pour les raisons illustrées dans la section 5.2.1, nous ne pouvons pas écrire les nouvelles écritures toute de suite après la fin du dernier enregistrement valable, mais nous devons plutôt les faire dans un nouveau secteur. Nous devons donc déterminer deux pointeurs de fin de journal :

- Le pointeur sur le dernier enregistrement valide, pour la récupération.
- Le pointeur sur le début du premier secteur non utilisé pour les nouveaux enregistrements

L'algorithme illustré par la figure 5.10 détermine ces deux pointeurs. De plus, il instaure la cassure dans le nouveau secteur, et rétablit le chaînage. La cassure est d'abord marquée en mémoire, dans la variable `IndCassure`. Pareillement, le pointeur de chaînage est d'abord écrit dans le tampon. L'écriture sur disque se fera quand le tampon sera plein.

5.4 Multiplexage des JLL sur un JP

Un journal physique peut contenir des enregistrements appartenant à des journaux logiques différents. L'ensemble des enregistrements appartenant à un journal donné, et stockés dans un journal physique donné s'appelle un *journal logique local (JLL)*. Le gestionnaire des journaux logiques locaux est responsable de multiplexer plusieurs journaux logiques locaux sur une journal physique unique. Chaque enregistrement porte dans son en-tête l'identificateur du JLL auquel il appartient.

Lors de l'écriture du journal, le gestionnaire des journaux logiques locaux s'occupe de l'ordonnancement des écritures, de manière à conserver une bande passante maximale, et de manière à garantir une latence minimale pour les requêtes *synchrones*. Les requêtes *synchrones* sont des requêtes qui doivent être confirmées à

```
Procédure BornesJournal2 : TypeCurseur ;
   Curseur ← BornesJournal1() ;

   /*localisation du pointeur fin pour les nouveaux
    *enregistrements */
   CurseurÉcriture ← ProchainSecteur(Curseur) ;
   IndCassure ← vrai ;

   /* localisation du pointeur fin pour la récupération */
   CursFin ← LireEntier(Curseur - 8) ;
   CursDébutDernier ← LireEntier(CursFin - 8) ;
   ÉcrireEntier(CursDébutDernier) ;

   retour CursFin ;
```

FIG. 5.10 – *Algorithme d'initialisation de pointeurs de fin*

l'application. Il s'agit là des requêtes de préparation, de validation, d'annulation et de synchronisation explicite.

Afin de pouvoir traiter en priorité les requêtes synchrones, tout en respectant l'ordre des enregistrements de données au sein de chaque journal logique, le gestionnaire de JLL gère une file d'attente pour chaque JLL. Quand il choisit le prochain enregistrement à écrire, il choisit de préférence un JLL qui contient un enregistrement correspondant à une requête synchrone, ainsi il garantit une latence minimale pour ce genre de requêtes. Quand il n'y a plus de file d'attente contenant des requêtes urgentes, il dessert les autres files, en remplissant de cette manière le secteur du journal physique qui contient l'enregistrement synchrone. Ainsi, en cas d'utilisation soutenue, il n'est pas nécessaire d'avoir recours à un remplissage artificiel, et nous conservons donc un haut débit. C'est seulement quand toutes les files sont vides qu'il devient nécessaire de « chasser » le dernier enregistrement synchrone par des zéros.

Afin de pouvoir localiser de manière efficace les JLL contenant une requête urgente, ceux-ci sont regroupés dans une liste chaînée.

5.5 Éclatement des JL en JLL

Le gestionnaire des journaux logiques (GJL) est responsable d'éclater le journal logique sur les différents sites de stockage participant. Chaque enregistrement est journalisé sur le même site que celui où réside l'image permanente du segment auquel il s'applique. Le GJL gère aussi la liste des sites de stockage participant à la transaction et place cette liste dans l'enregistrement de préparation. Il joue le rôle d'un *coordonnateur* lors de la validation du journal. Lorsque le PJS envoie l'ordre de valider un journal, le GJL donne d'abord l'ordre à tous les sites participant de préparer leur journal. Dès que l'enregistrement de préparation est sur le disque, les sites

de stockage envoient la confirmation de l'opération au GJL. Dès que le GJL a reçu ces messages de confirmation de tous les sites, il procède à la phase de validation. Dès que tous les sites ont confirmé la seconde phase, le GJL confirme l'opération à l'application[19].

5.6 Gestion de l'obsolescence

Afin de pouvoir mettre à jour la position du début du journal, nous devons savoir, à tout moment, quelle est l'emplacement dans le journal du plus ancien enregistrement non répercuté. Afin de faire ceci, le GJP gère une file d'attente d'enregistrements, qui est triée par position dans le journal. Étant donné que le journal est circulaire, on ne peut pas le trier sur la valeur numérique de la position telle quelle. On trie plutôt selon la quantité non signée *position_de_l_enregistrement* − *position_du_début_du_journal*. Cet ordre de tri n'est pas modifié pour les petits mouvements du début du journal. Dans ce contexte, un mouvement est «petit» s'il ne «croise» pas d'«enregistrements valables» (cf. figure 5.11).

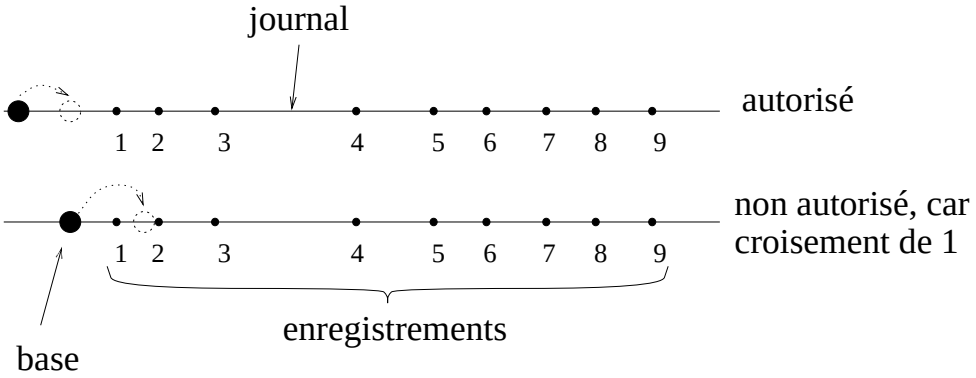

FIG. 5.11 – *Base de comparaison pour les positions des enregistrements*

La file d'attente est stockée sous la forme d'un tas (*heap*) ([AHU83]). Un tas est un arbre binaire qui a la propriété que la valeur de chaque nœud est inférieure à la valeur de chacun de ces fils. La valeur de la racine est donc le minimum du tout. Dans notre cas, il s'agit du pointeur sur le début du journal. L'insertion et l'enlèvement d'un élément dans un tas est de complexité $log(n)$, où n est la taille maximale jamais atteinte du tas (qui correspond à la quantité maximale d'enregistrements non répercutés). Ce coût est suffisamment faible pour nous permettre de gérer des quantités importantes d'enregistrements.

5.6.1 Enlèvement d'un élément du tas

Avant de pouvoir enlever un enregistrement portant un identificateur donné du tas, nous devons d'abord connaître son adresse. Une table de hachage accompagnant l'arbre nous permet de faire cette opération.

19. Théoriquement, nous pourrions confirmer l'opération de validation à l'application dès l'achèvement de la préparation. Mais malheureusement cette optimisation compliquerait le mécanisme d'obsolescence.

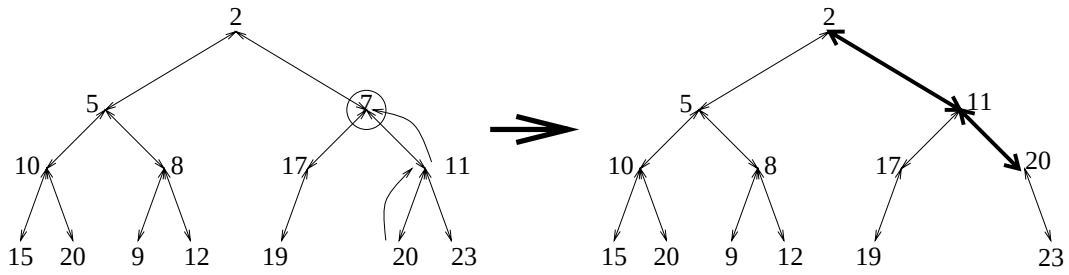

FIG. 5.12 – *Enlèvement d'un élément du tas*

Comme l'illustre la figure 5.12, après avoir enlevé l'enregistrement du tas, il reste un trou à cet endroit. Nous comblons ce trou en y mettant le fils de valeur la plus faible. Dans ce cas, il faut que nous nous occupions du nouveau trou ainsi formé. Dans le cas d'un nœud unaire, nous remontons toute la portion de l'arbre inférieur d'un cran (cf. figure 5.13), dans le cas d'une feuille nous nous arrêtons simplement. La figure 5.14 décrit l'algorithme d'enlèvement de feuille.

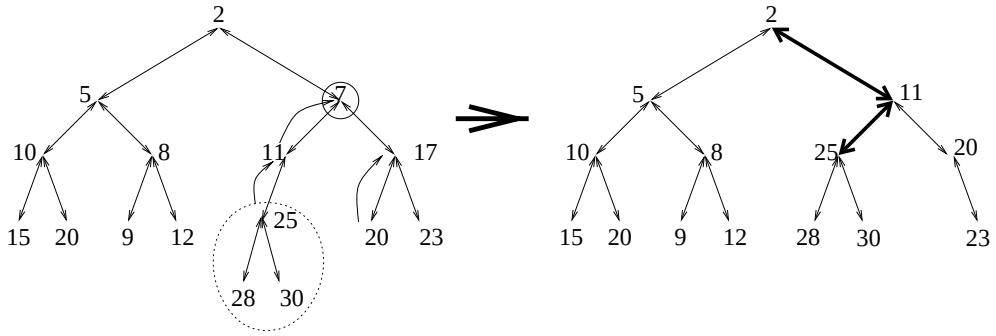

FIG. 5.13 – *Enlèvement d'un élément du tas avec nœud unaire*

5.6.2 Insertion d'un nouvel élément dans un tas

Les performances d'insertion et d'enlèvement en $log(n)$ ne peuvent être garanties que si le tas reste à peu près équilibré. Nous devons donc insérer les nouveaux éléments de préférence dans les sous-arbres les plus légers. C'est pourquoi nous gardons sur chaque nœud un indicateur d'équilibre qui représente la différence de poids entre les deux sous-arbres. Cet indicateur doit être mis à jour lors de chaque enlèvement ou ajout.

Les figures 5.15 et 5.16 illustrent le procédé d'insertion d'un nouvel élément dans le tas.

Cet algorithme d'insertion nous garantit que la profondeur après une insertion ne va jamais excéder $log_2 n$, où n est le nombre d'éléments dans l'arbre. Cependant, l'enlèvement d'éléments de l'arbre peut le déséquilibrer grièvement dans certains cas. Donc le nombre n utilisé pour le calcul de la complexité de l'algorithme n'est pas le nombre actuel de nœuds dans l'arbre, mais sa taille maximale qui ait jamais été atteinte. La taille maximale de l'arbre reste cependant bornée par la taille physique du journal.

```
Procédure EnlèveNœud(Tas : TypeTas, Nœud : PointeurNœud) ;
   DébutBoucle
      si NœudBinaire?(Nœud)
         si Valeur(FilsD(Nœud)) < Valeur(FilsG(Nœud)) alors
            Echange(Nœud, FilsD(Nœud))
            Nœud ← FilsD(Nœud) ;
            Équilibre(Nœud) ← Équilibre(Nœud) - 1 ;
         sinon
            Echange(Nœud, FilsG(Nœud))
            Nœud ← FilsG(Nœud) ;
            Équilibre(Nœud) ← Équilibre(Nœud) + 1 ;
         fin si
      sinon si NœudUnaire?(Nœud)
         /* nous remontons le sous-arbre fils d'un cran */
         DéplaceVers(Fils(Nœud), Nœud) ;
      sinon /* le nœud est une feuille */
         retour ;
      fin si
   FinBoucle
```

FIG. 5.14 – *Enlèvement d'une feuille du tas*

5.6.3 Maintien des indicateurs d'équilibre

Lors de chaque opération d'insertion ou d'enlèvement, nous devons mettre à jour les indicateurs d'équilibre de certains nœuds. Il s'agit là des nœuds qui se trouvent sur le chemin qui mène de la racine de l'arbre jusqu'à la feuille insérée ou enlevée. Lors d'une insertion, nous faisons les mises à jour nécessaires à chaque fois que nous déplaçons le curseur. Pour les opérations d'enlèvement, nous avons en plus besoin de parcourir le chemin qui mène de l'ancien emplacement de l'enregistrement enlevé à la racine. Cette opération n'est pas encore illustrée dans la figure 5.14, mais elle l'est dans 5.17.

Dans les dessins d'arbres qui précèdent, les chemins le long desquels les indicateurs d'équilibre doivent être mis à jour sont représentés en gras sur l'arbre résultat

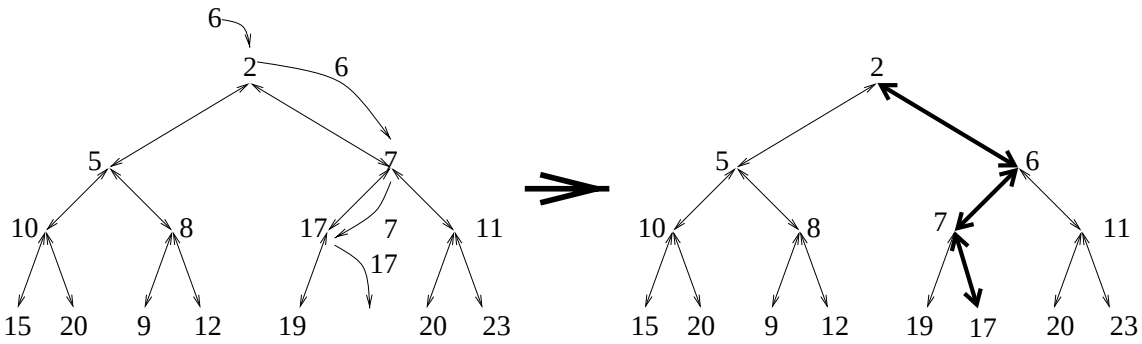

FIG. 5.15 – *Insertion d'un élément dans un tas*

```
Procédure InsèreNœud(
              Tas : TypeTas,
              NouvNœud : PointeurNœud) : PointeurNœud ;
   Nœud ← Racine(Tas) ;
   DébutBoucle
      si Valeur(NouvNœud) < Valeur(Nœud) alors
         Échange(NouvNœud, Nœud) ;
         /* NouvNœud est maintenant le nœud qui était
          * initialement dans l'arbre, avant l'échange */
      fin si
      si FilsD(Nœud) ← vide alors
         Équilibre(Nœud) ← Équilibre(Nœud) + 1 ;
         FilsD(Nœud) ← NouvNœud ;
      sinon si FilsG(Nœud) ← vide alors
         Équilibre(Nœud) ← Équilibre(Nœud) - 1 ;
         FilsG(Nœud) ← NouvNœud ;
      sinon /* nœud binaire */
         si Équilibre(Nœud < 0 alors
            /* Le côté gauche est plus lourd, allons à droite */
             * pour compenser */
            Nœud ← FilsD(Nœud)
         sinon
            Nœud ← FilsG(Nœud)
         fin si
      fin si
   FinBoucle
```

FIG. 5.16 – *Algorithme d'insertion d'un nœud dans un tas*

(côté droit).

Comme la longueur du chemin vaut au plus la profondeur de l'arbre, la complexité de la mise à jour des indicateurs est aussi en $logn$.

5.7 Récupération après une panne

5.7.1 Objectifs et modèle de récupération après une panne

Dans Arias, la récupération après panne se fait enregistrement par enregistrement. Pour chaque enregistrement à refaire (ou à défaire), le service de journalisation générique appelle la routine de récupération du PJS responsable pour le JL auquel l'enregistrement appartient.

En concevant notre méthode de récupération nous avons accordé une grande importance à sa capacité de fonctionner en mode dégradé si la situation l'impose. Par exemple, nous voulons pouvoir récupérer le résultat d'une transaction même si

```
Procédure EnlèveNœud2(Tas : TypeTas, Nœud : PointeurNœud) ;
   tant que Nœud ≠ Racine(Tas) faire
      NouvNœud ← Père(Nœud) ;
      si FilsD(NouvNœud) ← Nœud alors ;
         Équlibre(NouvNœud) ← Équlibre(NouvNœud) - 1 ;
      sinon
         Équlibre(NouvNœud) ← Équlibre(NouvNœud) + 1 ;
      fin si
      Nœud ← NouvNœud
   fin tant que
```

FIG. 5.17 – *Mise à jour des indicateurs d'équilibre entre la racine et le nœud enlevé*

les autres sites qui ont participé à cette transaction sont inaccessibles. Comme nous le verrons dans la suite, ceci n'est pas possible dans tous les cas (zone grise). Dans ces cas, nous voulons être capables de pouvoir récupérer les autres transactions, qui ne souffrent peut-être pas de ce problème elles-mêmes, et ceci tout en respectant des dépendances éventuelles entre les transactions. De plus, même en présence de transactions non récupérées, nous permettons le relancement des applications si celles-ci n'utilisent pas de segments affectés par ces transactions.

5.7.2 Protocole d'accord réparti

Comme décrit plus haut, nous utilisons un protocole de validation à deux phases. Celui-ci nous permet, dans la plupart des cas de déterminer l'état d'une transaction après une panne, même si les autres sites ne sont pas joignables. Un journal logique est validé s'il contient un enregistrement de préparation et de validation. Il est annulé soit s'il contient un enregistrement d'annulation explicite, soit s'il contient uniquement des enregistrements de données, sans enregistrement de préparation.

Cependant, si le journal contient un enregistrement de préparation, mais pas d'enregistrement de validation, son état ne peut être déterminé à partir des seules informations locales. On dit que le journal logique est dans un *état indéfini*. On peut aussi dire qu'il est dans la *zone grise*. Dans ce cas, nous devons essayer de contacter les autres sites, et attendre leur réponse.

5.7.3 Mise en attente de journaux logiques et de segments

Si, lors de la récupération, nous trouvons un JLL qui est dans la zone grise, et dont les autres sites participants sont inaccessibles, il serait dommage de retarder pour autant la récupération du journal physique tout entier. Dans un cas pareil, nous arrêtons, dans un premier temps, juste la récupération du JLL en question tout en continuant de récupérer les autres.

Cependant cette approche pose le problème d'éventuelles dépendances de données entre les différentes transactions. En effet, des modifications effectuées sur des mêmes données doivent êtres rejouées dans le même ordre que lors de l'opération initiale.

Or avec notre algorithme de mise en attente, il devient possible que des journaux logiques écrits récemment soient récupérés avant des journaux logiques plus vieux, si ces vieux journaux sont dans la zone grise et attendent la réponse d'un site couramment inaccessible.

Afin de résoudre ce problème, nous mettons aussi les données en attente. La récupération de toute donnée touchée par un journal est mise en attente jusqu'à ce que l'on connaisse l'état du journal. Afin de limiter l'espace mémoire des structures de données utilisées par le récupérateur, nous groupons ensemble toutes les données appartenant au même segment. Si un journal en attente (on dit aussi *bloqué*) touche à une donnée appartenant à un certain segment, nous bloquons tout le segment.

La chaîne des dépendances s'arrête là. En effet, ce n'est pas la peine de bloquer un journal logique tout entier juste parce qu'il contient des enregistrements bloqués, étant donné que les enregistrements sont autonomes, et donc indépendants à condition qu'ils touchent à des zones distinctes.

Si à la fin de la récupération, il reste des segments bloqués, l'accès à ces segments est refusé aux applications, tandis que les autres segments sont accessibles normalement. Une application essayant d'accéder un segment bloqué reste bloquée jusqu'à ce que le segment soit récupéré.

5.7.4 Les déroulement de l'algorithme de récupération

Nous utilisons un algorithme à trois passes :

- La première passe est une passe de «reconnaissance». Son but est de déterminer quels sont les journaux qui sont localement validés, abandonnés ou dans l'état indéfini. Le journal physique est lu à l'envers (des enregistrements les plus récents aux enregistrements les plus vieux) afin de voir les enregistrements décrivant l'état des journaux logiques en premier.

- À la fin de première passe, commence la «négociation» avec les pairs (autres sites), afin de déterminer l'état définitif des journaux qui sont dans l'état indéfini sur le site local. La communication avec les autres sites permet d'enlever l'ambiguïté au sujet de l'état de la plupart de ces journaux. Cependant, on lance la seconde passe même si on ne connaît pas encore l'état de tous les journaux.

- Durant la seconde passe, le journal est parcouru à l'endroit, c'est-à-dire des enregistrements les plus anciens vers les enregistrements les plus nouveaux. Cette passe rejoue les enregistrements des *journaux après* qui sont validés.

- Durant la troisième passe, le journal est parcouru à l'envers. Cette passe défait les enregistrements des *journaux avant* qui sont abandonnés.

Comme nous interdisons toute dépendance entre les journaux après et les journaux avant (en exigeant qu'un segment donné ne peut être couvert que par des journaux avant ou par des journaux après), les deux passes peuvent se dérouler en parallèle. Le sens d'évolution des ces deux passes n'est qu'une indication générale. En effet il y a des retours en arrière à chaque fois que l'état d'un journal précédemment indéfini devient connu.

5.7.5 Les structures de données utilisées

Le récupérateur maintient, pour chaque journal logique figurant dans le journal physique et pour chaque segment modifié par un enregistrement, une structure de données décrivant leur état.

L'adresse des ces structures peut être trouvée rapidement grâce à une table de hachage indexée par le numéro du segment ou du journal logique respectivement.

La structure décrivant l'état du segment contient entre autres les champs suivants :

- Deux curseurs `Position` qui pointent vers le dernier enregistrement décrivant ce segment qui soit récupéré. L'un est pour la seconde passe, et l'autre pour la troisième passe,
- Un indicateur `IsBlocked` qui indique si le segment est bloqué ou non,
- Des pointeurs de chaînage `next` et `prev` qui permettent de ranger les descripteurs de segment dans une liste chaînée basée sur le journal logique à cause duquel ils sont bloqués

La structure JLL contient entre autres les éléments suivants :

- La base `First` de la liste des segments bloqués sur ce journal,
- L'état du journal tel qu'il est connu à l'instant actuel,
- La liste des participants à la transaction décrite par ce journal, si elle est connue (cette liste peut être initialisée à partir de l'enregistrement de préparation),
- Une liste de sites qui sont d'accord avec nous sur l'état du journal. Cette liste est utilisée pour promouvoir un journal de l'état indéfini vers l'état validé s'il s'avère qu'il est indéfini sur tous les sites participant

De plus, le récupérateur utilise deux curseurs globaux (communs à tous les JLL) qui repèrent la position des prochains enregistrements à récupérer (pour la seconde et la troisième passe respectivement).

5.7.6 La gestion des journaux et segments bloqués

Pour chaque enregistrement le récupérateur déroule les tests et opérations suivantes :

1. Si le segment touché par l'enregistrement a déjà était récupéré, on passe à l'enregistrement suivant. On peut constater ceci en inspectant le curseur pointant sur le dernier enregistrement récupéré. Cette situation peut se produire après le recul du curseur de récupération lors du déblocage d'un journal.
2. Si le segment est bloqué, on passe à l'enregistrement suivant.

3. Si le journal logique auquel appartient l'enregistrement est dans un état indéfini, on le bloque en mettant son indicateur de blocage à 1, et en le plaçant dans la liste chaînée attachée au journal en question. Puis on passe à l'enregistrement suivant.

4. Si nous arrivons à ce point- ci, cela signifie que nous avons toutes les données nécessaires pour traiter l'enregistrement. Nous avançons donc le curseur du segment concerné. Le traitement de l'enregistrement se déroule de la manière suivante :

   - Si nous sommes dans la passe 2 et que le JLL auquel appartient l'enregistrement est validé, nous l'envoyons au PJS pour qu'il le rejoue,
   - Si nous sommes dans la passe 3 et que le JLL auquel appartient l'enregistrement est abandonné, nous l'envoyons au PJS pour qu'il le défait,
   - Dans les autres cas, l'enregistrement ne nécessite pas d'action de la part du PJS, et nous passons à la suite.

Dès que l'état d'un journal devient défini, nous parcourons la liste chaînée des segments bloqués sur lui. Pour chacun de ces segments, nous positionnons l'indicateur de blocage à zéro, et nous reculons le curseur de la passe courante vers le curseur du journal à débloquer, si celui-ci est plus ancien.

5.7.7 Le protocole d'accord entre sites

Le protocole d'accord se déroule en plusieurs étapes. Avant la première passe, un site A qui revient après une panne envoie des messages de présence aux autres sites. Par ces messages, le site A annonce à ses pairs qu'il est désormais opérationnel. Les autres sites réagissent en lui envoyant des messages qui décrivent l'état des journaux logiques auquel A a participé.

Aucun traitement des messages reçus n'a lieu pendant la première passe. En effet durant cette passe, le système ne connaît même pas encore l'état local. Tout message reçu pendant cette passe est donc mis en attente, et traité une fois l'état local connu.

5.7.7.1 Le format des messages

Il y a deux sortes de messages :

- Les messages de présence de type `HELLO` contiennent uniquement l'identité de l'expéditeur.

- Les messages de type `LOGSTATE` servent à communiquer à d'autres sites l'état des journaux. Il sont constitués d'une suite de triplets (`journal, état, liste_de_sites`). La liste de sites énumère les sites qui sont au courant de l'état tel qu'il est listé dans ce message. Cette liste sert à détecter la situation où tous les participants à une transaction voient le journal dans un état indéfini. Par convention, on déclare le journal comme validé dans une telle condition.

5.7.7.2 Traitement d'un message

Quand un site B reçoit un message de type `HELLO` de la part d'un site A il y répond en envoyant un message de type `LOGSTATE` à A, qui contient l'état de tous les journaux logiques indéfinis dans lesquels A et B ont participé. Éventuellement, ce message peut-être vide (c'est-à-dire juste constitué d'un en-tête). Cette réponse joue deux rôles :

- Il indique au site A que B est opérationnel. En effet, A, qui revient juste de redémarrer, n'a probablement pas vu les messages `HELLO` en provenance de B, étant donné que ceux-ci ont été ont envoyés probablement quand A n'était pas encore opérationnel. C'est pour cette raison que ce message doit être envoyé même s'il s'avère être vide,
- Il demande au site A des informations sur les journaux logiques indéfinis listés.

Quand un site A reçoit un message de type `LOGSTATE` de la part d'un site B, il le passe en revue, enregistrement par enregistrement :

- Si le champ **état** correspond à l'état tel qu'il est connu localement, la liste de sites contenue dans le message est rajoutée à la liste de sites locale.
- Si le champ **état** est indéfini, plusieurs cas peuvent se présenter :
  - Si le journal a un état défini en local sur A, A va envoyer cet état à B dans un message de réponse,
  - Si le journal est dans un état indéfini sur A, A vérifie, grâce à la liste des sites, s'il est indéfini partout. Si oui, le journal est promu validé, et B est informé de ce fait.

6

Évaluation du système

6.1 Environnement d'essai

Pour réaliser nos mesures, nous n'avons pas utilisé de module de journalisation spécifique (voir 4.1.3), mais nous avons écrit une application d'essai qui envoie ses messages directement à la couche générique de journalisation (voir 4.2). Cependant, à part le module spécialisé, notre pile de streams Arias (voir A.2) contient tous les modules, y compris le multiplexeur, et les modules de cohérence.

Notre application d'essai mesure le temps nécessaire à l'écriture et à la validation de journaux, ainsi que le temps nécessaire à certaines étapes. Elle permet de faire varier le nombre de journaux, le nombre d'enregistrements et la taille des enregistrements.

6.2 Quantités mesurées

À l'issue de chaque expérience, quatre temps sont imprimés :

1. Le temps total pris par l'opération toute entière,
2. Le temps pris par la création des journaux,
3. Le temps pris par les préparations,
4. Le temps pris par les validations.

Nous avons constaté que seul le temps pris par l'opération toute entière, ainsi que le temps pris par la préparation varient de manière appréciable, donc seuls ces derniers temps ont été représentés. Pendant la préparation, les service générique de journalisation doit écrire tous les enregistrements qui sont encore en attente dans le journal à valider. Le temps de préparation reflète entre autre la taille de la file d'attente des enregistrements (cf. 5.4) au moment de la préparation. Une partie du temps de préparation est aussi due à l'aller-retour des messages.

Pour avoir des temps plus longs et donc plus facilement mesurables, nous créons pour chaque point de mesure dix journaux de composition identique (taille et nombre d'enregistrements). Les temps montrés dans les graphes sont les temps pris par l'écriture de ces dix journaux. Afin de couvrir une plage de valeurs assez large, nous

doublons la variable analysée à chaque pas de mesure. Pour la taille des enregistrements, nous retranchons 16 de la « valeur ronde » afin de tenir compte de l'en-tête qui est rajouté par le service de journalisation.

La plupart de nos graphiques contiennent aussi des mesures « de référence » qui ont été faites avec une pile de streams vide (c'est-à-dire sans service de stockage), sur laquelle nous avons envoyé le même nombre de messages de la même taille que pour nos mesures « utiles ». Ces mesures servent à dégager le surcoût dû à la communication par streams et par réseau.

6.3 Paramètres explorés

Nous avons réalisé 6 expériences :

1. La première expérience vise à mesurer la latence de préparation et de validation. Elle consiste à créer 10 journaux contenant très peu de données (10 enregistrements de 10 octets). Son but est uniquement de mesurer la latence de la validation. L'expérience est faite sur un site.

2. Les expériences qui vont suivre visent à mesurer la bande passante de journalisation.

   La seconde expérience consiste à choisir une taille d'enregistrement moyenne (environ 500 octets), et de faire varier le nombre d'enregistrements.

3. Pour la troisième expérience, nous faisons l'inverse : Nous choisissons un nombre d'enregistrements moyen (environ 500), et nous faisons varier leur taille.

4. Les deux expériences précédentes nous montrent que l'impact de la taille des enregistrements est moins important que l'impact du nombre. La quatrième vise à étudier l'impact du regroupement des données sur le débit. Pour tous les points de mesure, une quantité de données constante est écrite dans le journal, mais ces données sont regroupées de manière différente à chaque fois : de nombreux petits enregistrements ou peu de grands enregistrements. Nous verrons que la performance dépend du groupement des données, même si la quantité totale de données reste la même.

5. La cinquième expérience reprend la quatrième expérience, mais sur une pile Arias munie uniquement des modules de stockage (donc sans le multiplexeur, et les modules de cohérence). Elle vise à illustrer l'influence du nombre de modules streams à traverser.

6. La sixième expérience vise à étudier l'impact du parallélisme des transactions sur les performances. Elle consiste à ordonnancer les opérations de telle manière que plusieurs journaux sont écrits en parallèle. Nous cherchons à mettre en évidence deux effets : tout d'abord, le fait qu'il y a plusieurs transactions parallèles permet de remplir les segments contenant des enregistrements synchrones par des données des autres transactions, plutôt que par des zéros inutiles. Ce phénomène devrait donc améliorer le débit vers le disque. Mais d'un autre côté, ce

parallélisme devrait consacrer une certaine partie de la bande passante à l'écriture des journaux moins urgents, laissant ainsi une file d'attente plus longue au journal à valider. Nous devrions donc observer un allongement des files d'attente (et donc du temps de préparation).

Les expériences 2, 3 et 4 ont aussi été réalisées en réparti sur deux machines. Entre l'expérience en centralisé et l'expérience en réparti correspondante, nous avons gardé le même nombre d'enregistrements. Donc en réparti, chacun des sites a eu à traiter deux fois moins d'enregistrements qu'en centralisé.

Chaque graphique comporte aussi des courbes « de référence » illustrant la partie du coût qui est due à la transmission des messages. Elles ont été obtenues en faisant transiter le même nombre de messages de même taille à travers la pile streams que pour les vraies expériences, mais en utilisant des modules vides à la place des modules de journalisation.

6.4 Résultats et interprétation

6.4.1 Latence de validation d'une transaction courte

Pour cette première expérience, nous avons mesuré le temps pris pour écrire et valider 10 journaux de 10 enregistrements chacun, où chaque enregistrement est composé de 10 octets de données utilisateurs.

L'expérience complète dure 360 ms, et le temps de préparation des 10 journaux est de 180 ms, soit 18 ms par journal.

Ces performances semblent très prometteuses. En effet, cela correspond à 1600 transactions par minute.

6.4.2 Influence du nombre d'enregistrements journalisés

Cette expérience 6.1 illustre l'influence du nombre des enregistrements, en gardant leur taille constante (496).

Le temps total (dans les deux cas) est presque proportionnel au nombre d'enregistrements. Comme l'illustrent les courbes de référence, ce temps est dû pour la majeure partie au coût des communications.

Le temps de préparation évolue de manière intéressante. Il croit légèrement pour l'expérience en centralisé, ce qui traduit un allongement de la file d'attente pendant l'expérience.

En réparti, ce temps est constant : le débit arrivant est suffisamment faible pour que la taille de la file d'attente atteigne un régime stationnaire. De plus, il est nettement plus faible que celui mesuré en réparti : ceci semble indiquer que la contribution la plus importante du coût de préparation est due au drainage de la file d'attente plutôt qu'aux temps d'aller-retour des messages de synchronisation.

6.4.3 Influence de la taille des enregistrements

Cette expérience illustre l'influence de la taille des enregistrements, en gardant leur nombre constant (512).

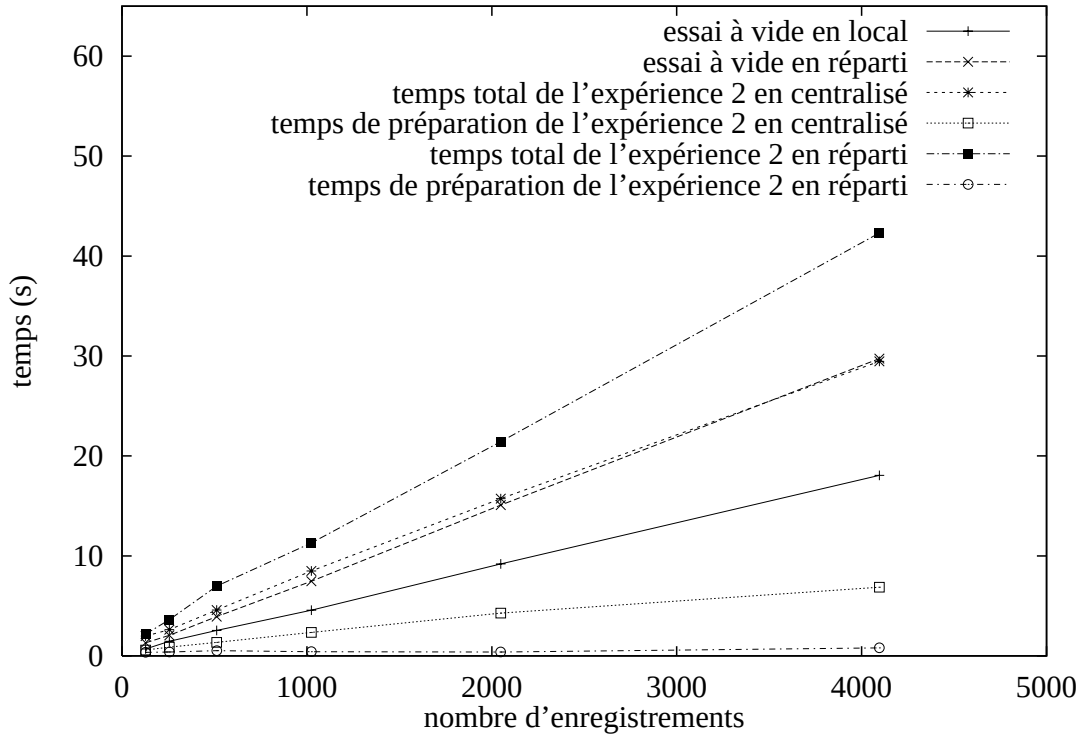

FIG. 6.1 – *Influence du nombre des enregistrements*

Ce graphique est assez délicat à interpréter.

Commençons par les mesures de référence. À l'exception de quelques légères perturbations, le temps d'exécution de l'expérience à vide en local reste constant. Ceci peut s'expliquer par le fait que les messages streams sont communiqués par *référence* plutôt que par *valeur*, c'est-à-dire que juste un pointeur est passé de module en module, sans copie de données (cf. A.1.5).

L'expérience à vide en réparti montre une dépendance plus marquée de la taille du message transmis. Ceci est normal, compte tenu du fait que les données doivent être physiquement envoyées sur un réseau à débit fini. Il est intéressant de constater que l'envoi de messages de 256 octets demande plus de temps que l'envoi de messages de 512 octets. Probablement, le protocole TCP/IP garde les petits messages dans un tampon en attendant la disponibilité d'une quantité de données plus grande avant de l'envoyer.

Le temps total de l'expérience « à charge utile » en centralisé croit avec la taille des enregistrements, ce qui est normal. Cependant, la partie du coût qui est indépendante de la taille est assez importante. Cette partie du coût peut probablement être imputée à la gestion des messages streams.

Le temps de préparation reste négligeable jusqu'à une certaine valeur seuil (entre 512 et 1024 octets). Probablement, en deçà de cette valeur, les données peuvent être vidées plus vite sur le disque qu'ils n'arrivent, et donc la file ne s'allonge pas. Au delà de ce seuil, la file s'allonge proportionnellement au débit entrant.

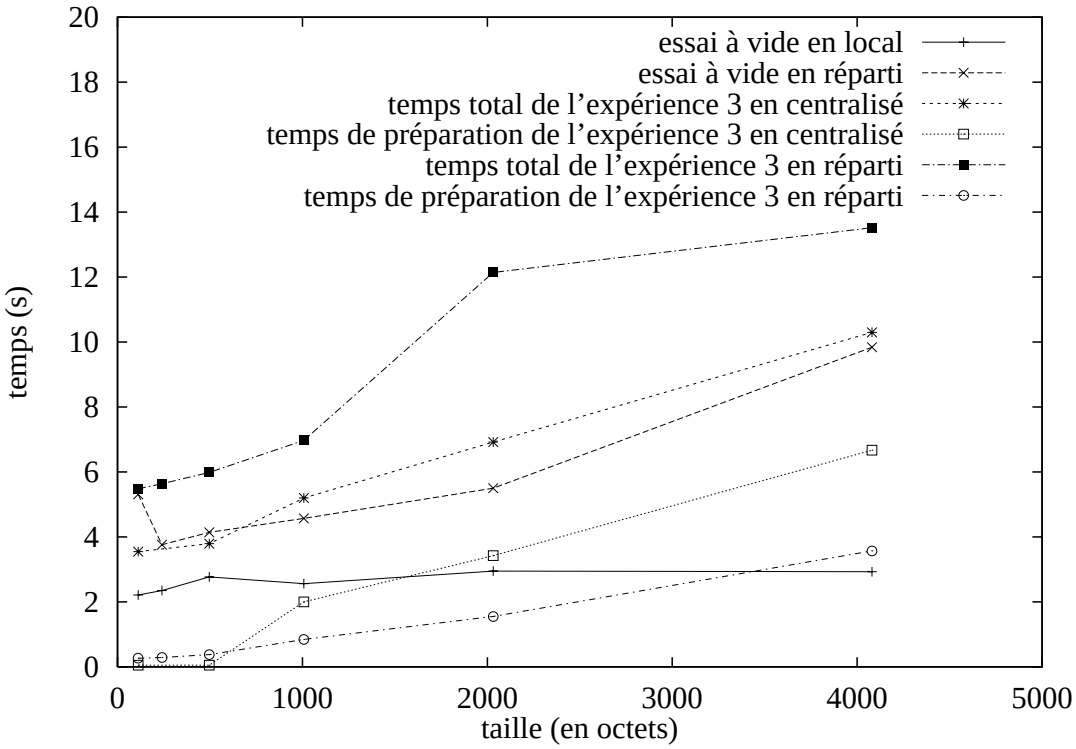

FIG. 6.2 – *Influence de la taille des enregistrements*

Le temps total de l'expérience en réparti est croissant en fonction de la taille. Tout comme l'expérience en centralisé, le coût fixe (c'est-à-dire la partie du coût qui est indépendant de la taille) est appréciable.

Le temps de préparation est lui aussi croissant, et ceci dès les petites tailles. Cependant, il croit moins vite que le coût de préparation en centralisé. Ceci est probablement dû au fait que le réseau agit comme goulot d'étranglement placé en amont des files d'attente, qui croissent donc moins. À première vu, il peut sembler paradoxal que la croissance de cette courbe commence dès le début, mais ceci s'explique probablement par la file d'attente de messages dans le pilote réseau du site expéditeur, qui doit elle aussi être drainée lors de la préparation. Comme le débit du réseau est plus faible que le débit du disque, cette file commence à se remplir plus tôt.

6.4.4 Influence du groupement des données

La figure 6.3 illustre notre quatrième et notre cinquième expériences. Les deux expériences consistent à écrire 1 Moctets de données sur le disque, en faisant varier le nombre d'enregistrements (et donc aussi leur taille).

Cette expérience montre que notre système a de meilleures performances si les données lui sont présentées sous forme de peu d'enregistrements de grande taille plutôt que sous forme de beaucoup d'enregistrements de petite taille. Nos courbes

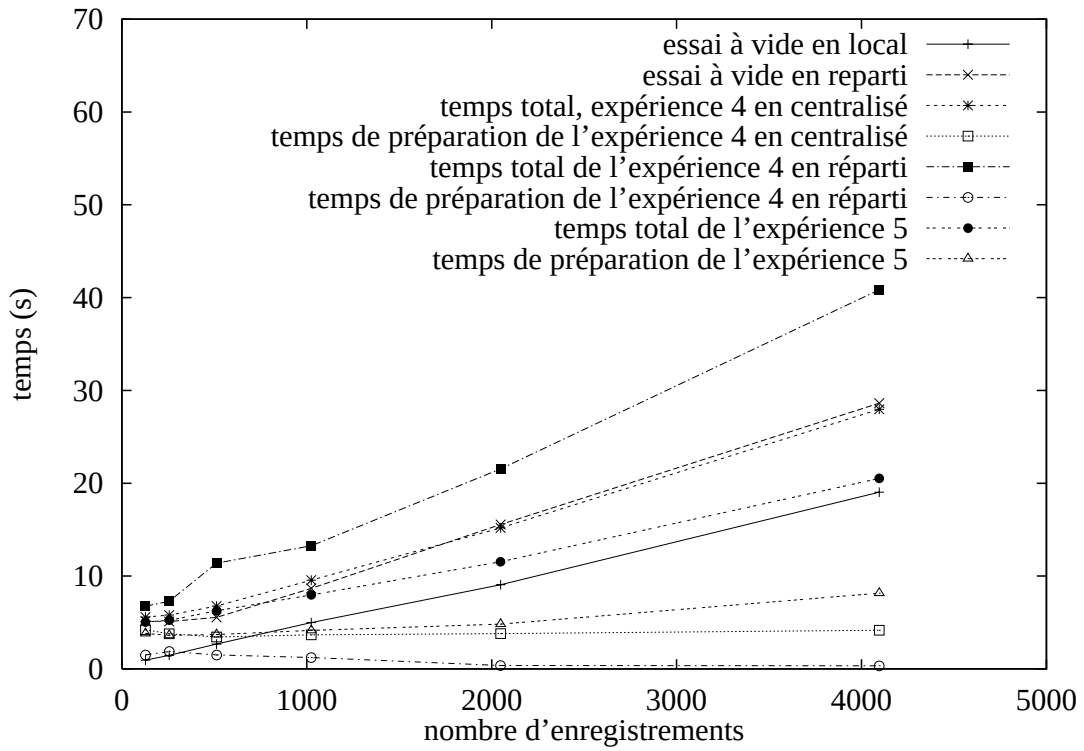

FIG. 6.3 – *Influence du groupement des données*

de références montrent que cette évolution est en large partie due au mécanisme de communication.

L'évolution du temps de préparation est intéressante : les temps de préparation les plus longs correspondent aux temps totaux les plus courts, et vice-versa. La raison de ce phénomène est que plus le temps total est long, plus le débit est faible, et donc les files d'attente des journaux croissent moins. C'est pourquoi la préparation est rapide, étant donné qu'il n'y a que peu de données à drainer.

En effet, nous voyons que (pour cette expérience), les temps d'exécution totaux les plus longs correspondent à des temps de préparation plus courts (donc moins de données en attente).

6.4.5 Comportement d'une pile streams allégée

La figure 6.3 illustre aussi notre expérience sur pile streams « allégée ». Les performances sont meilleures qu'avec la pile de streams Arias complète. Ceci montre donc que malheureusement nous payons la souplesse de notre système, qui nous oblige à multiplier les modules présents dans la pile streams, par des performances moins bonnes. Une nouvelle version du système qui utilise moins de modules différents est en cours de développement.

Un autre détail intéressant de l'expérience est que maintenant le temps de préparation et le temps total varient dans le même sens. Ce qui se passe probablement,

c'est que maintenant les données arrivent plus rapidement dans la couche de journalisation qu'elles ne peuvent être écoulées. Cependant, étant donné que la variation se produit bien que la quantité de données totales soit constante, le goulot d'étranglement se situe avant les routines d'écriture sur disque (mais après le traitement des files d'attente). C'est probablement les appels aux routines de manipulation des messages streams qui nous coûtent cher.

6.4.6 Influence du parallélisme des transactions

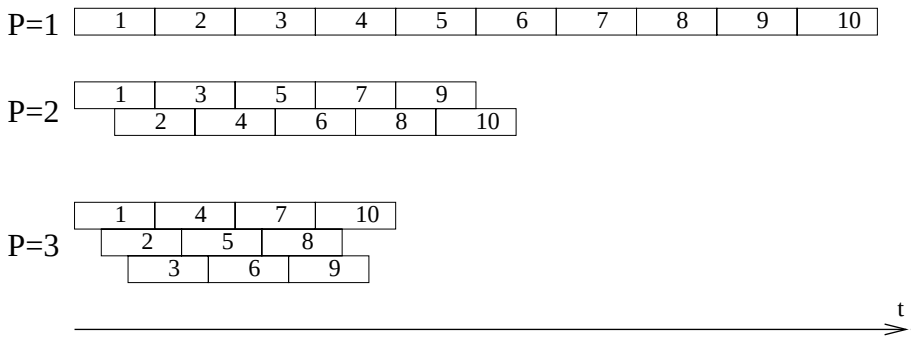

FIG. 6.4 – *Ordonnancement de l'écriture des journaux*

Cette expérience consiste à journaliser une quantité identique de données, mais d'ordonnancer les journaux de manière différente. Sans parallélisme, les journaux sont écrits les uns après les autres. Avec un degré de parallélisme de 2, nous commençons à écrire le second journal quand le premier est écrit à moitié. À partir de cet instant l'écriture des deux journaux se fait en parallèle. Quand le premier journal, qui a eu une avance sur le second, est écrit entièrement le troisième commence à être écrit, et ainsi de suite. La figure 6.4 illustre cet ordonnancement d'écriture des journaux.

Le but de cette expérience est de voir si le débit des écritures est augmenté par le parallélisme. En effet le parallélisme devrait permettre d'éviter les pertes dues au remplissage des secteurs de fin de journal par des zéros. Nous réalisons deux séries de mesures en faisant varier le degré de parallélisme de 1 à 3. Les deux séries utilisent des journaux à 4096 enregistrements. Dans la première série nous utilisons des enregistrements « longs » de 240 octets, alors que dans la seconde nous utilisons des enregistrements courts de 10 octets.

La figure 6.5 montre que l'influence du parallélisme est quasiment nulle pour les journaux à enregistrements courts. Pour les journaux à enregistrements longs, nous constatons une augmentation du temps d'exécution total et du temps de préparation. L'allongement du temps de préparation est probablement dû au fait qu'avant la préparation, la moitié (ou le tiers) de la bande passante vers le disque est accaparée par des enregistrements non urgents, ce qui mène à un allongement de la file d'attente du journal à préparer. Nous constatons que les courbes du temps de préparation et du temps total sont parallèles : c'est donc l'allongement du temps de préparation à lui seul qui contribue à cette légère perte de performances.

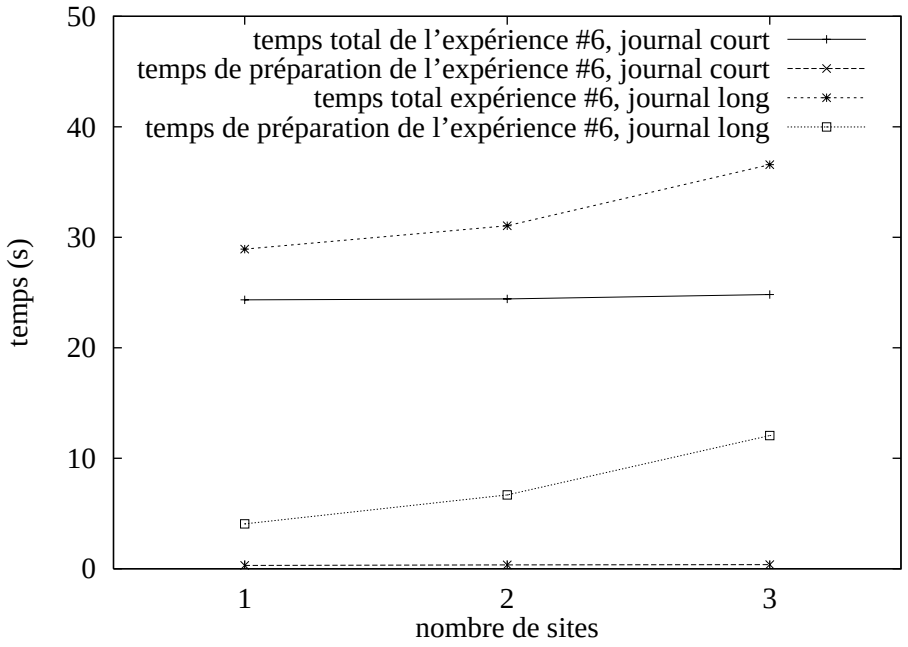

FIG. 6.5 – *Influence du parallélisme des transactions*

6.5 Conclusion

Nous avons constaté dans la première expérience que dans des conditions favorables nos performances sont proches de celles permises par le matériel.

La seconde expérience (nombre d'enregistrements variable) nous a montré que le coût de la manipulation des messages streams en local, qui dépend surtout de leur nombre et non de leur taille, est loin d'être négligeable.

Les résultats de la troisième expérience (taille variable) sont plus durs à interpréter à cause des nombreux facteurs intervenants. La présence de nombreuses files d'attente qui se déversent les unes dans les autres, et qui sont présentes à de multiples endroits du système, complique l'analyse des événements. D'éventuelles recherches ultérieures seront nécessaires pour mieux comprendre les phénomènes présents afin d'améliorer les performances.

La quatrième expérience (groupement) nous confirme que notre système a de meilleures performances quand il traite peu de grands messages que quand il traite beaucoup de petits messages. Étant donné que le coût de manipulation des messages streams dépend surtout du nombre de messages manipulés, et moins de leur taille, ceci suggère une certaine inefficacité dans la gestion de nos messages streams. Des recherches ultérieures seront nécessaires pour confirmer nos hypothèses, pour trouver la source de ce problème et pour le corriger.

La cinquième expérience (groupement, sur pile allégée) met plus directement en cause les streams : en effet, en allégeant notre pile streams, les performances s'améliorent. Suite à cette constatation, nous avons donc décidé de munir certains des modules d'une interface procédurale directe afin de court-circuiter le mécanisme

streams pour des messages entre des modules génériques locaux. Une amélioration de notre gestion du contrôle de flux pourrait nous permettre de mieux optimiser l'écoulement des messages dans les files.

La sixième expérience (influence du parallélisme des transactions) montre que les performances de notre système face à des transactions parallèles restent bonnes, surtout si nous utilisons des enregistrements de petite taille.

Dans toutes ces expériences, nous constatons que les temps totaux en réparti sont supérieurs à ceux en centralisé, mais la différence reste dans les limites du raisonnable (performances de 20% à 50% inférieures au centralisé). Les temps de préparation des expériences en réparti (qui interviennent dans la latence de l'opération de validation) sont même souvent **inférieurs** à ceux de l'expérience correspondante en centralisé. Ceci est dû à des raisons de contrôle de flux.

Nous voyons donc que la puissance et la souplesse de notre système ne sont pas obtenues au détriment des performances.

7

Conclusion

Le travail présenté dans cette thèse relève du domaine de la mise en œuvre de systèmes de stockage pour bases de données. La conception de notre système est issu de la volonté de fournir une interface plus souple aux applications gérant des bases de données. En effet, de nombreux SGBD n'utilisent pas les primitives du système d'exploitation sous-jacent, étant donné que celles-ci ne correspondent pas exactement aux besoins.

Nous avons donc mis en œuvre notre service de stockage comme une boîte à outils offrant de fonctions de base pour la mise à jour atomique des données stockées. Les mécanismes sont mis en œuvre dans les couches basses de notre système, tandis que les applications fournissent des modules qui mettent en œuvre la politique.

Dans la suite de cette conclusion, nous allons rappeler la motivation de notre projet, ensuite nous allons repasser en revue les principaux aspects de notre mise en œuvre, puis nous mettons en relief les contributions de notre travail. La quatrième section fait une synthèse de l'évaluation des performances de notre système, et la cinquième section donne des perspectives et ouvertures.

7.1 Aspects fonctionnels

Les systèmes de bases de données mettent souvent en œuvre leur propre gestion du stockage, en court-circuitant le système d'exploitation sous-jacent. Une raison pour ceci est que la plupart des systèmes d'exploitation n'offrent pas *exactement* l'interface que les gestionnaires de bases de données attendent. Nous avons donc choisi de mettre en œuvre un système *souple* qui offre aux applications une grande liberté quant au choix des politiques de *synchronisation* et de *journalisation*. Ainsi nous espérons attirer les concepteurs de SGBD, qui trouveront beaucoup d'avantages dans notre approche :

- Les données sont directement couplées en espace utilisateur, et il est donc possible d'y accéder de manière plus simple et efficace ;

- Les problèmes de double pagination inhérent à beaucoup de systèmes SGBD mis en œuvre entièrement en mode utilisateur sont éliminés ;

- Le système fournit un support pour la gestion de la cohérence de la mémoire. Cette gestion est faite *en connaissance de cause* grâce à l'interaction avec la gestion de la synchronisation,

Par rapport aux systèmes traditionnels de mémoire virtuelle répartie, nous fournissons :

- La résistance aux pannes. La nécessité d'intégrer la mémoire virtuelle avec la gestion de la fiabilité a été déduite d'une analyse des systèmes existants de mémoire virtuelle, et des systèmes existants de stockage, ainsi que de l'observation de leur limites.

- La souplesse. Les modules spécifiques de cohérence et de synchronisation permettent au gestionnaire de la cohérence de la mémoire vive de prendre ses décisions en connaissance de cause et ainsi d'éviter les transferts inutiles de données entre machines. Les modules spécifiques de journalisation permettent de fournir aux applications le niveau de fiabilité dont elles ont besoin, en évitant ainsi de les faire payer (en perte de performances) pour un service dont elles n'ont pas besoin.

7.2 Mise en œuvre

Cette thèse décrit une approche répartie pour la mise en œuvre d'un système à mémoire virtuelle récupérable. Pour cette mise en œuvre, nous avons utilisé les mécanismes suivants :

- L'efficacité de la mémoire virtuelle est obtenue par l'intégration entre la gestion de la cohérence, de la synchronisation et du stockage ;

- Nous utilisons un journal de type *Write Ahead*. Ce journal assure l'atomicité des mises à jour, et il permet également la mise à jour de l'image permanente des segments en différé, et donc en dehors du chemin critique ;

- Nous entrelaçons les enregistrements du journal des différentes transactions, afin d'optimiser l'utilisation de la bande passante vers le disque. Nous pouvons donc remplir les blocs devant être écrits de manière synchrone (préparation, validation, etc.) par des enregistrements d'autres transactions, plutôt que par des zéros.

- La souplesse de notre système est obtenue en découpant le service de stockage en deux couches : une couche spécifique fournie par l'application qui lui permet d'utiliser son propre protocole de journalisation spécifique et une couche générique qui s'occupe uniquement du stockage physique, et de l'accord entre sites. L'interprétation du contenu du journal est à la charge de l'application.

- La répartition existe à la fois côté client (une transaction peut être composée d'agents se déroulant sur plusieurs sites) que côté serveur (une transaction peut manipuler, de manière atomique, des données qui ont leur lieu de résidence sur différents sites)

7.3 Contributions

Cette thèse illustre la faisabilité d'un système souple à mémoire virtuelle répartie récupérable. Afin de réaliser une mise en œuvre pratique, plusieurs structures de données et algorithmes ont dû être développés. Les contributions de cette thèse sont résumés ici :

- La mise en œuvre d'une mémoire virtuelle répartie résistante aux pannes utilisable pour des gestionnaires de bases de données orienté objet,
- La description des algorithmes nécessaires pour interfacer le stockage avec la mémoire virtuelle, et pour mettre en œuvre physiquement ce service de stockage :
  - Le protocole de communication entre le sous-système de journalisation et le sous-système de pagination, qui permet notamment la mise à jour différée de l'image stable, et la purge du journal au moment où les modifications décrites ont effectivement été réalisées en dur,
  - Le format du journal physique, qui permet d'obtenir un bon débit tout en maintenant une latence faible,
- Une analyse qui a permis l'intégration de la mémoire virtuelle avec le service de stockage,
- Des évaluations de la performance de notre mémoire virtuelle répartie résistante aux pannes.

7.4 Évaluation

Les premiers résultats sont encourageants. Pour les transactions courtes, nous obtenons un débit d'environ 1500 transactions par minute. Pour les transaction longues, à grand volume de données, nous obtenons un débit de 6 Moctets journalisés par minute.

Les difficultés rencontrées lors de la réalisation ont été plutôt d'ordre technique que conceptuel. En effet, la programmation et la mise au point de programmes qui se déroulent en mode noyau sont toujours délicates.

Après coup, les streams étaient un bon choix, en ce qui concerne la souplesse. En effet, ce mécanisme nous a permis de développer les différentes parties du système de manière indépendante et de les intégrer facilement. Cependant, l'inconvénient de cette approche est une légère perte de performances due aux délais occasionnés par le passage des requêtes dans la pile des streams. En effet, nous avons constaté que l'envoi de 10000 messages sur une pile de streams vide prend 4 secondes. Nous envisageons donc de sortir certains des modules de la pile de streams et de les intégrer plutôt par une interface procédurale directe.

Ce travail a prouvé que, malgré le surcoût dû au mécanisme streams, nous avons obtenu des performances très bonnes à la fois en ce qui concerne la latence des validations et le débit de données vers le journal.

7.5 Perspectives

Le travail dans lequel nous nous sommes engagés est un travail de grande ampleur, car il couvre la mise en œuvre d'un système complet, avec ses aspects cohérence et synchronisation, pagination, protection et finalement fiabilité du stockage. Maintenant que la couche système est établie, il reste à fournir des modules spécialisés, des exécutifs (librairies), et des applications.

Afin de valider notre approche, nous envisageons une coopération avec l'équipe d'O2, afin de voir ce que les mécanismes de cohérence de mémoire et de journalisation que nous fournissons dans Arias peuvent apporter à leur SGBDOO.

Arias a été conçu pour s'exécuter au-dessus des architectures à 64 bits d'adresse. Cependant, étant donné que des versions à 64 bits d'adresse de AIX ne sont pas disponibles dans le futur immédiat, notre première version d'Arias tourne sur 32 bits. Malgré cette restriction initiale, la conception et le codage ont été faits en considérant tous les problèmes et perspectives associés aux grands espaces d'adressage, et le portage sur la plate-forme définitive pourra donc être fait facilement, une fois qu'elle sera disponible.

Dans certaines situations, les performances ont souffert à cause de l'utilisation des streams comme support de communication entre modules. Il serait intéressant de comparer le coût de cette mise en œuvre utilisant des streams avec une mise en œuvre utilisant un autre support.

Notre système a été conçu dès le départ avec le but d'offrir une interface souple et personnalisable. Il nous est donc pas possible actuellement d'étudier le coût de cette souplesse, car elle fait partie intégrante du système. L'étude de ce coût pourra faire l'objet de recherches futures.

La gestion de journaux et de volumes répliqués est envisageable. Ceci nous permettra d'assurer une plus grande disponibilité des données permanentes, au prix d'une gestion de l'atomicité un peu plus compliquée.

Nous considérons aussi une nouvelle mise en œuvre de notre gestionnaire de volume, qui ne s'appuierait plus sur le JFS de AIX, et gagnerait ainsi en performance en éliminant le fonctions inutilisés de JFS et leur coût.

Bibliographie

[1st91] 1st ed. MIPS Computer Systems, Inc., Sunnyvale, California. « *MIPS R4000 Microprocessor User's Manual* », 1991.

[ADL90] M. Ahamad, P. Dasgupta et R. J. LeBlanc. « Fault-tolerant Atomic Computations in an Object-based Distributed System ». *Distributed Computing*, 4(2), mai 1990.

[AHU83] Alfred Aho, John Hopcroft et Jeffrey Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading MA, 1983.

[BAC+90] Haran Boral, William Alexander, Larry Clay, George Copeland, Scott Danforth, Michael Franklin, Brian Hart, Marc Smith, et Patrick Valduriez. « Prototyping Bubba, a Highly Parallel Database System ». *IEEE Transactions on Knowledge and Data Engineering*, 2(1), mars 1990.

[BAHK+88] J.M. Bernabéu-Aubán, P.W. Hutto, M.Y.A. Khalidi, M. Ahamad, R.J. LeBlanc, W.F. Appelbe, P. Dasgupta, et U. Ramachandran. « The Architecture of Ra : A Kernel for Clouds ». Rapport Technique CIT-ICS-88/25, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, 1988.

[BCZ91] J. K. Bennet, J. B. Carter et W. Zwaenepoel. « *Munin : Distributed Shared Memory Using Multi-Protocol Release Consistency* », volume 563 de *Lecture Notes in Computer Science*, pages 56–60. Springer Verlag, 1991.

[BHPT93] J. Besancenot, L. Hammami, P. Pucheral, et J.M. Thévenin. « GEODE, a multi-threaded storage object library for advanced transactional applications ». Rapport Technique 93.39, Masi, Paris, 1993.

[BHPT94] J. Besancenot, L. Hammami, P. Pucheral, et J.M. Thévenin. « A low level storage manager supporting the development of advanced systems ». Dans *Proc. of the 3rd Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI sponsored by IFIP, IEEE and AFCET)*, Rabat, Morocco, avril 1994.

[BTR78] V. Berthis, C. D. Truxal et J. G. Ranwailer. « *System/38 Addressing and Authorisation* », pages 24–31. IBM, 1978.

[BZ91] B. Bershad et M. Zekauskas. « Midway : Shared Memory Parallel Programming with Entry Consistencyy for Distributed Memory Multiprocessors ». Rapport Technique, Carnegie-Mellon University, septembre 1991.

[CD89] R.C. Chen et P. Dasgupta. « Linking Consistency with Object/Thread Semantics : An Approach to Robust Computation ». Dans *Ninth International Conference on Distributed Computing Systems (IC-DCS)*, pages 121–129, juin 1989.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, et Micheal J. Zwilling. « Shoring Up Persistent Applications ». Dans *23 ACM SIGMOD Conf. on the Management of Data*, pages 383–392, juin 1994.

[Che92] Pierre Yves Chevalier. « A Replicated Object Server for a Distributed Object-Oriented System ». Dans 11^{th} *Symposium on Reliable Distributed Systems*, Houston, TX, octobre 1992.

[CLBHL92] J.S. Chase, H. M. Levy, M. Baker-Harvey, et E. D. Lazowska. « Opal : A Single Address Space System for 64-Bit Architectures ». Dans *Proceedings of the Third IEEE Workshop on Workstation Operating Systems*. IEEE, 1992.

[CLFL93] J.S. Chase, H.M. Levy, M.J. Feeley, et E.D. Lazowska. « Sharing and Protection in a Single Address Space Operating System ». Rapport Technique 93-04-02, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, avril 1993.

[CLLBH92a] J.S. Chase, H. M. Levy, E. D. Lazowska, et M. Baker-Harvey. « Lightweight Shared Objects in a 64-Bit Operating System ». Dans *7th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, octobre 1992.

[CLLBH92b] J.S. Chase, H.M. Levy, E.D. Lazowska, et M. Baker-Harvey. « Lightweight Shared Objects in a 64-Bit Operating System. ». Dans *Proceedings of the 7th Annual Conference on Object-Oriented Programming Systems, Languages and Applictions (OOPSLA '92)*, pages 397–413, Vancouver, British Columbia, Canada, octobre 1992. ACM Press.

[Dan88] D. S. Daniels. « *Distributed Logging for Trancation Processing* ». PhD thesis, Carnegie-Mellon Univ., Pittsburgh, PA, décembre 1988. Tech. Rep. CMU-CS-89-114.

[DCM+91] P. Dasgupta, R. C. Chen, S. Menon, M.P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R.J.

LeBlanc, W.F. Appelbe, et J.M. Bernabéu-Aubán. « The Design and Implementation of the Clouds Distributed Operating System ». Rapport Technique, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, 1991.

[Dig92] Digital Equipment Corporation, Maynard, Mass. « *Alpha Architecture Handbook* », 1992.

[DLAR91] P. Dasgupta, R.J. LeBlanc, M. Ahamad, et U. Ramachandran. « The Clouds Distributed Operating System ». Rapport Technique, College of Computing, Georgia Tech, Atlanta, 1991.

[DO89] F. Douglis et J. K. Ousterhout. « Log-Structured File Systems ». Dans *Intellectual leverage, COMPCON Spring 89, Thirty-fourth IEEE Computer Society International Conference*, San Francisco, CA, Février Mars 1989.

[DST87] D. S. Daniels, A. Z. Spector et D. S. Thompson. « Distributed Logging for Transaction Processing ». Dans *Proc. of the ACM SIGMOD Int. Conf.*, volume 16, pages 82–96, San Francisco, CA, mai 1987.

[Epp89] Jeffrey Eppinger. « *Virtual Memory Management for Transaction Processing Systems* ». PhD thesis, Carnegie Mellon Univ., février 1989.

[FC87] R. S. Finlayson et D. R. Cheriton. « Log Files : An Extended File Service Write-Once Storage ». Dans *Proc. of the Eleventh ACM Symposium on Operating Systems Principels*, volume 21, pages 139–148, Austin, TX, novembre 1987. ACM.

[Fin89] R. S. Finlayson. « *A Log File System Exploiting Write-once Storage* ». PhD thesis, Stanford University, Palo Alto, California, juillet 1989. Rapp. Tech. Stan-Cs-89-1272.

[FZT+92a] Michael J. Franklin, Michael J. Zwilling, C. K. Tan, Michael J. Carey, et David J. DeWitt. « Crash Recovery in Client-Server EXODUS ». Dans *SIGMOD Conference*, pages 165–174, 1992.

[FZT+92b] Michael J. Franklin, Michael J. Zwilling, C. K. Tan, Michael J. Carey, et David J. DeWitt. « Crash Recovery in Client-Server EXODUS ». Dans *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, San Diego, CA, juin 1992.

[Gif79] D. K. Gifford. « Weighted voting for replicated data ». Dans *Proceedings of the Seventh ACM Symposium on Operating System Principles*, pages 150–159, Pacific Grove, CA, décembre 1979.

[GMB+81] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, et Irving Traiger. « The Recovery Manager of the System R Database Manager ». *Computing Surveys*, 13(2), juin 1981.

[GR93] Jim GRAY et Andreas REUTER. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA 94403, 1993.

[Gra78] Jim GRAY. « *Notes on Data Base Operating Systems* ». Lecture Notes in Computer Science. Springer Verlag, New York, 1978.

[Ham95] L. HAMMAMI. « *Une boîte à outils pour la construction et la fédération de multiples sources de données dans un système distribué* ». Thèse de Doctorat, Université Pierre et Marie Curie, Paris VI, Paris (France), février 1995.

[HERV93] G. HEISER, K. ELPHINSTONE, S. RUSSELL, et J. VOCHTELOO. « Mungi : a Distributed Single Address-Space Operating system ». Rapport Technique, School of Computer Science and Engineering, University of New South Wales, novembre 1993.

[HR83] T. HAERDER et A. REUTER. « Principles of Transaction Oriented Database Recovery - A Taxonomy ». *ACM Comput. Surv.*, 15(4), décembre 1983.

[MSSW94] Cathy MAY, Ed SILHA, Rick SIMPSON, et Hank WARREN, éditeurs. *The PowerPC Architecture : A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.

[MSWK93] K. MURRAY, T. STIEMERLING, T. WILKINSON, et P. KELLY. « Angel : Resource Unification in a 64-bit Micro Kernel ». Rapport Technique, Department of Computing, Imperial College Longon, juin 1993.

[Org72] E. I. ORGANICK. *The Multics System : An Examination of its Structure*. MIT Press, Cambridge, Mass., 1972.

[Par93] Craig PARTRIDGE. *Gigabit Networking*. Addison-Wesley, Reading MA, 1993.

[RHB+90] J. ROSENBURG, F. HENSKENS, F. BROWN, R. MORRISON, et D. MUNNO. « Stability in a Persistent Store Based on a Large Virtual Memory ». Rapport Technique CS/90/6, Department of Mathematical and Computational Sciences, University of St. Andrews, 1990.

[RO91] M. ROSENBLUM et J. K. OUSTERHOUT. « Design and Implementation of a Log Structured File System ». Dans *Proc. of the Thirteenth ACM Symp. on Oper. Syst. Principles*, volume 25, pages 1–15, Pacific Grove, CA, octobre 1991.

[Ruf92a] Michel RUFFIN. « Kitlog : a Generic Logging Service ». Dans *Proc. of the Eleventh Symposium on Reliable Distributed Systems*, pages 139–146, Houston, TX, octobre 1992.

[Ruf92b] Michel RUFFIN. « *Kitlog : Un Service de Journalisation Générique* ». Thèse de Doctorat, Université Pierre et Marie Curie, Paris VI, Paris (France), septembre 1992. TU-0205, ISBN-2-7261-0759-1.

[Ruf95] Michel RUFFIN. « Issues in Logging Techniques through a Study of Four Systems ». Rapport Technique, Broadcast, février 1995.

[SKW92] Vivek SINGHAL, Sheetal V. KAKKAD et Paul R. WILSON. « Texas : An Efficient, Portable Persistent Store ». Dans *Proc. Fifth Int'l. Workshop on Persistent Object Systems*, San Miniato, Italy, septembre 1992.

[Sol95] Franck G. SOLTIS. *Inside the AS/400*. Duke Press, Loveland, Colorado, 1995.

[SZ90] E. SHEKITA et M. ZWILLING. « Cricket : A Mapped Persistent Object Store ». Dans *Proc. of the Persistent Object Systems Workshop*, Martha's Vineyard, MA, septembre 1990.

[Tho79] R. H. THOMAS. « A majority consensus approach to concurrency control for multiple copy databases ». *ACM Transactions on Database Systems*, 4(2) :180–209, juin 1979.

[WD94] Seth J. WHITE et David J. DEWITT. « QuickStore : A High Performance Mapped Object Store ». Dans *SIGMOD Conference*, pages 395–406, 1994.

[Wil91] P. R. WILSON. « Pointer Swizzling at Page Fault Time : Efficiently Supporting Huge Address Spaces on Standard Hardware ». *Computer Architecture News*, 19(4) :6–13, 1991.

[Y+87] M. YOUNG et OTHERS. « The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System ». Dans *Proc. of the 11th Symposium on Operating System Principles*, novembre 1987.

A

Interface entre Arias et AIX

Ce chapitre décrit comment nous utilisons les mécanismes fournis par AIX pour mettre en œuvre notre système.

A.1 Communication par stream

La communication entre les différents modules de Arias, et entre les différentes instances de ces modules se fait par les *streams*. Comme l'illustre la figure A.1, un stream est un canal de communication bidirectionnel au sein du noyau qui relie un processus utilisateur à un pilote d'entrée-sortie stream.

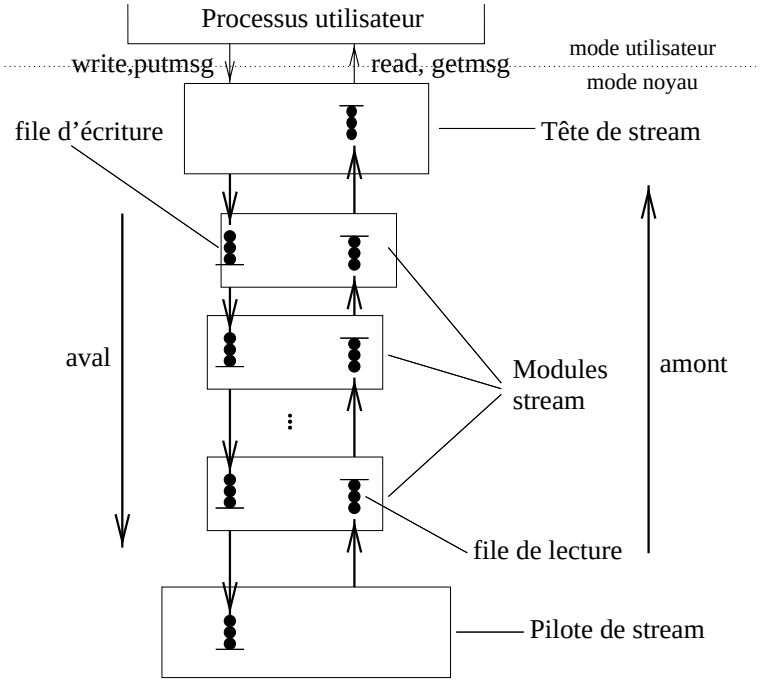

FIG. A.1 – *Pile de* stream

A.1.1 Les briques de base d'un stream

Les composants primaires d'un stream sont la *tête* de stream, le pilote de stream et éventuellement des *modules* de stream qui se trouvent entre la tête et le pilote. Le tout constitue une *pile* stream. Une pile stream ressemble à un *pipeline* en shell, à ceci près que les données vont dans les deux sens.

A.1.1.1 Tête de stream

Dans une pile stream, la tête de stream est le bout qui fournit l'interface entre le stream et le processus utilisateur. Les fonctions principales de la tête de stream sont :

- Le traitement des appels systèmes spécifiques aux stream.
- Le transfert d'informations entre le processus utilisateur et le stream.

A.1.1.2 Le pilote de stream

Dans un stream, le pilote fournit l'interface entre un périphérique et le stream. Le pilote peut aussi être un *pseudo*-pilote qui n'est pas directement associé à un périphérique. C'est le cas du pilote stream que nous avons mis en œuvre pour Arias. Il est aussi possible de réaliser des *multiplexeurs* qui permettent des bifurcations dans la pile stream. Le multiplexeur est vu comme un pilote par les modules qui sont au-dessus de lui.

A.1.1.3 Les modules stream

Un module stream est une entité qui contient des routines de traitement pour les données qui y transitent. Un module est toujours placé au milieu de la pile, entre la tête et le pilote. Un module est l'équivalent stream des commandes shell dans un pipeline, à l'exception près qu'un module contient deux fonctions de traitements, une pour chaque direction. La direction de la tête vers le pilote est appelé *en aval*, tandis que la direction du pilote vers la tête est appelé *en amont*.

A.1.2 Les messages et les files

A.1.2.1 Messages

L'information circule le long du stream sous forme de *messages stream*. Un message *stream* est composé d'un ou plusieurs blocs de données, et des structures de contrôle associées.

A.1.2.2 Files d'attente

Chaque module de stream possède deux files d'attente dans lesquelles sont stockés les messages en attendant leur traitement. Les messages *en amont* sont stockés dans la *file de lecture* tandis que les messages *en aval* sont stockés dans la *file d'écriture*.

A.1.3 Les multiplexeurs

Comme l'illustre la figure A.2, un multiplexeur permet d'introduire des bifurcations dans une pile stream. Il existe plusieurs sortes de bifurcations :

- Plusieurs pilotes peuvent être reliés à un même processus utilisateur (côté gauche de la figure),
- Un pilote peut être lié à plusieurs processus utilisateurs (milieu),
- Plusieurs pilotes peuvent être liés à plusieurs processus utilisateurs (côté droit).

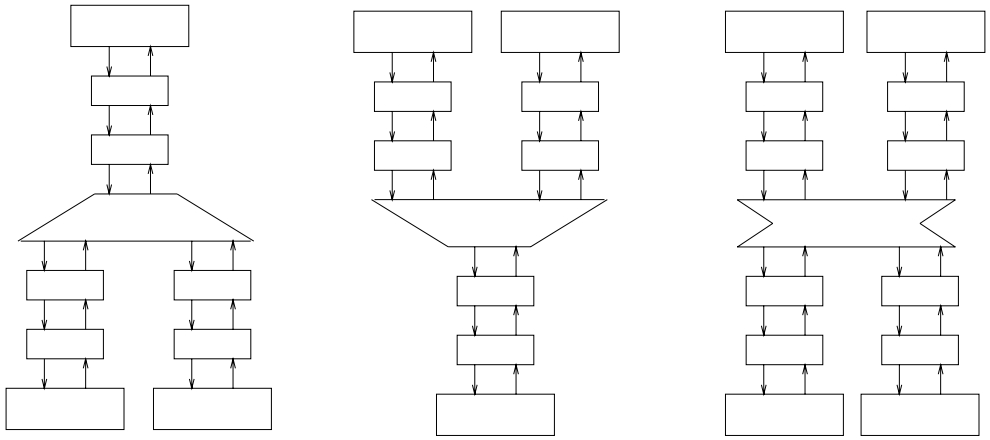

FIG. A.2 – *Utilisation de multiplexeurs*

Le système stream ne fournit pas de multiplexeurs en tant que tels, mais plutôt les primitives nécessaires pour en construire. En effet, la mise en œuvre d'un multiplexeur dépend grandement de la nature du routage qu'il est censé effectuer ; il n'est donc pas possible que le système puisse fournir des multiplexeurs par défaut.

A.1.4 Traitement d'un message par un pilote ou par un module

Quand un module stream reçoit un message, il peut réagir de la manière suivante :

- il peut le transmettre au prochain module,
- il peut y apporter quelques changements, et puis le transmettre,
- il peut le traiter, et puis y répondre, en le renvoyant dans la direction d'où il est venu,
- il peut le traiter sans y répondre.

Un module stream peut aussi émettre des messages de manière spontanée. Nous utilisons cette faculté pour générer les confirmations d'écriture faites par le service de stockage. En effet, ces confirmations ne sont pas déclenchées directement par l'arrivée d'un message stream, mais par un événement extérieur à la pile stream (à savoir par le déverrouillage du tampon d'entrée-sortie qui contient les données écrites).

A.1.5 Structure d'un message stream

Un message stream est une liste chaînée de blocs. Un bloc de données peut appartenir à plusieurs streams à la fois. Ceci permet d'optimiser le temps de copie des messages, et de gagner de la place en partageant des données entre plusieurs messages.

Afin de permettre ce partage, plusieurs indirections sont utilisées, comme l'illustre la figure A.3. Le message est constitué d'une liste chaînée de structures du type `msgb` (message bloc). La structure `msgb` elle-même n'est pas partageable entre messages, mais les données référencées par elle le sont. Chaque `msgb` comporte trois pointeurs permettant de retrouver les données du bloc :

- Le pointeur `b_rptr` pointe vers le début des données,
- Le pointeur `b_wptr` pointe vers la fin des données,
- Le pointeur `b_datap` pointe vers une structure de type `datab`.

La structure `datab` sert à gérer le partage des données. Elle comporte entre autres un compteur de références qui reflète le nombre de `msgb`s qui partagent les données, et un pointeur vers le début de la zone allouée pour les données. Ce pointeur n'est pas nécessairement égal au pointeur `b_rptr`, en effet un bloc de message n'est pas obligé d'utiliser tout l'espace alloué pour ses données. Ceci permet une certaine souplesse quant aux choix des données partagées. Dans notre exemple, le message du haut vaut «ci exempes si», composé à partir des trois morceaux «ci» «exemp» «es si». Pareillement, le second message vaut «is un exe xamp». Notons que le second message retient une sous-chaîne différente («un exe») du second bloc. L'indépendance entre les pointeurs `b_rptr` et `b_wptr` d'un côté, et des pointeurs contenus dans `b_datap` permettent cette souplesse.

La structure `datab` contient aussi un champ *type*, qui permet de distinguer entre autres entre blocs de *données* (`DATA`) et blocs de contrôle (`CONTROL`).

A.1.5.1 Mise à jour de messages

Cette organisation permet de mettre à jour facilement le message, sans avoir besoin d'en recopier des parties. On peut par exemple rajouter du texte au début sans avoir besoin de décaler le reste : il suffit en effet de rajouter un nouveau bloc.

Pareillement, grâce aux pointeurs de début de fin contenus dans chaque `msgb`, il est facile d'enlever des morceaux de texte, même si ceux-ci n'occupent pas un bloc entier.

À cause de ces deux propriétés, l'architecture stream est particulièrement adaptée à la mise en œuvre des protocoles de communication par réseau, qui nécessitent souvent l'ajout et l'enlèvement d'en-têtes au fur et à mesure que le message transite à travers les différentes couches.

A.1.5.2 Partage de parties arbitraires d'un message

Grâce au niveau d'indirection supplémentaire, un même `datab` peut être référencé par plusieurs blocs de messages. Afin de faciliter cet usage, la structure `datab`

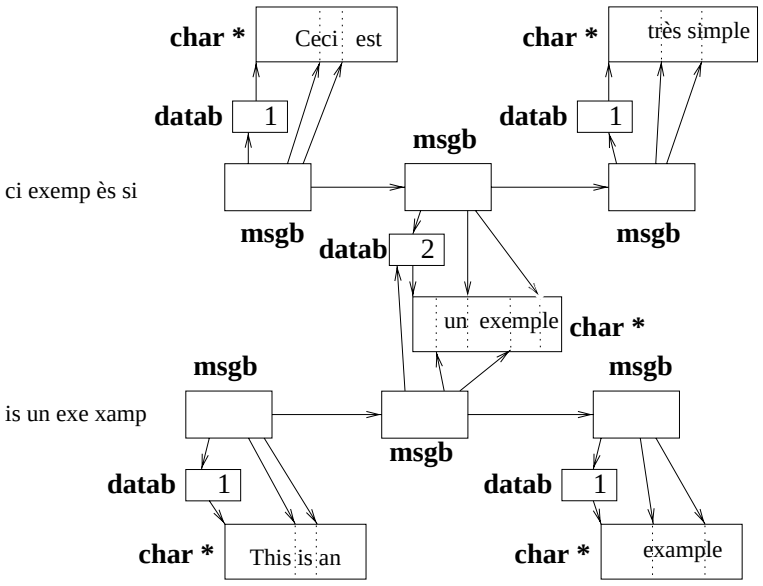

FIG. A.3 – *Structure des messages* stream

contient un compteur de références, qui est automatiquement tenu à jour par les primitives stream offerts par AIX.

Avant de modifier le contenu d'un bloc de message partagé, il faut cependant en faire une copie physique si on ne veut pas que les autres messages qui utilisent le même bloc soient modifiés aussi (*copy on write*)

A.1.5.3 Types associés au blocs de messages

Chaque bloc de message stream comporte un *type* représenté par un entier de 0 à 255. Ce type représente la nature du message et vaut le plus souvent `M_DATA`. Par convention, le type d'un message est le type de son premier bloc.

A.2 La pile de stream Arias

La figure A.4 montre la pile de stream Arias. On y distingue les modules suivants :

- Les têtes de stream (non représentées) servent d'interface entre Arias et ses applications.
- Le multiplexeur sépare les modules spécifiques à une instance d'application des modules communs à toutes les applications qui tournent sur un site.
- Le PJS/PCS contient les routines de journalisation et de récupération attachées au PJS ou PCS en question.
- Le module de protection agit comme un filtre, et refuse les messages qui essayeraient de faire des opérations non autorisées à l'expéditeur.
- Le module de journalisation s'occupe de la journalisation et de la reprise après panne.

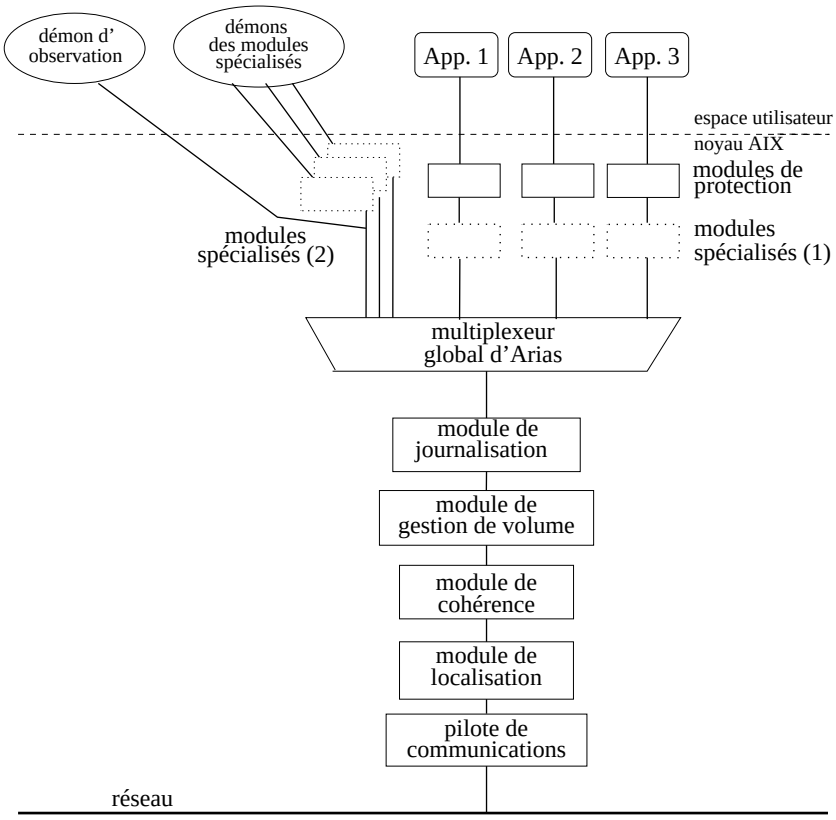

FIG. A.4 – *Pile stream de Arias*

- Le module de gestion de volume s'occupe du stockage permanent des données à long terme.

- Le module de cohérence gère la cohérence entre la mémoire vive des différents sites, et la synchronisation.

- Le module de localisation permet de déterminer le site sur lequel sont stockés certaines structures de contrôle vitales.

- Le pilote de communication réalise l'interface de la pile stream avec le réseau, et permet ainsi au différents sites de communiquer entre eux.

- Les démons de modules spécialisés servent à donner une pile de résidence aux instances des modules spécialisés chargés de la reprise. En effet, ces instances ne peuvent pas être stockées dans la piles streams d'une application, étant donné que lors de la reprise, les applications n'existent pas encore. Ces démons et les modules dans leur pile servent aussi à assurer la coordination entre les différentes instances d'un même protocole sur le site, et ils servent d'interlocuteur aux instances sur les autres sites, dans les cas où il n'y aurait pas d'application utilisant ce protocole sur ce site.

- Le démon d'observation sert à observer ce qui se passe dans le système.

A.3 Formats de messages utilisés dans Arias

Dans Arias, un message commence obligatoirement par un en-tête qui contient des informations de routage du message. Ces informations comprennent notamment :

- le destinataire,
- l'expéditeur (pour savoir où envoyer la réponse),
- une description de la zone de données concernée,

Cet en-tête est stocké dans un bloc de type *contrôle*, tous les blocs suivants sont stockés dans des blocs de *données*.

Chaque communicant (destinataire et expéditeur) est décrit par les informations suivantes :

- Le site sur lequel il réside,
- Le type *vertical* du module (journalisation, gestion du volume, gestion de la cohérence, module de protection, protocole spécialisé),
- L'instance particulière du module (le type *horizontal*), s'il s'agit d'un protocole spécialisé,
- La fonction à appeler dans le module destinataire (ou la fonction à appeler dans le module expéditeur au moment de la réception de la réponse). L'identificateur de la fonction chez le destinataire est aussi appelé le *type Arias* de ce message (À ne pas confondre avec le type stream du message).
- Une estampille permettant d'indentifier la réponse dans le cas où plusieurs requêtes entre les mêmes partenaires seraient en cours.

La description de la zone concernée contient l'adresse de départ et l'étendue de la zone. Cette information est surtout utile au module de protection, qui veille à ce que les applications ne modifient que les données autorisées, en filtrant les messages inappropriés. Évidemment, les autres modules utilisent cette même information pour réaliser leurs opérations. Il n'est donc pas possible à un pirate de contourner le système en envoyant un message où la description de la zone est en désaccord avec d'autres informations contenues dans le message.

A.3.1 Formats des messages pour le service de stockage

Les modules de stockage (volume, journal, PJS) utilisent un second en-tête qui contient les paramètres scalaires de la fonction à appeler. Il s'agit là notamment de positions d'enregistrements dans le journal, d'identificateurs de journaux et d'enregistrements etc.

Ensuite viennent les données elles-mêmes. Ce sont des données qui seront sauvegardées (ou lues) telles quelles sur disque : contenu de zones de données et enregistrements de journaux, avec ou sans en-têtes.

La figure A.5 montre une réponse à une requête de lecture de journal. On y distingue l'en-tête de routage, l'en-tête contenant les paramètres scalaires (début et fin de l'enregistrement), et finalement le contenu de l'enregistrement, avec son en-tête à lui (identification du journal, type d'enregistrement etc).

L'appendice B énumère les différents types de message stream concernant le stockage.

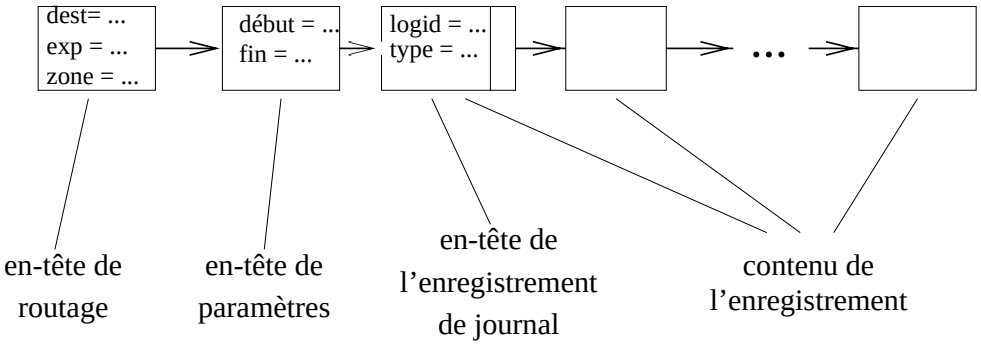

FIG. A.5 – *Exemple d'un message stream de Arias*

A.4 Environnements d'exécution

Comme tous les systèmes Unix modernes, AIX possède plusieurs environnements d'exécution dans le noyau, qui permettent chacun de faire certaines opérations, mais pas d'autres :

- L'environnement processus. Cet environnement permet le plus d'opérations. Il permet de faire des fautes de page, d'attendre sur des variables de condition et d'utiliser les verrous d'exclusion mutuelle. Du code s'exécutant dans cet environnement est requérable. Les processus noyau, et les processus utilisateur pendant un appel système se déroulent dans cet environnement.

- L'environnement stream. Les routines de traitement de messages associées aux différents modules et pilotes stream se déroulent dans cet environnement. Il permet de faire des fautes de page, mais il impose des limites assez fortes sur l'usage de la synchronisation. Du code s'exécutant dans cet environnement est requérable, mais le programmeur d'un module stream peut demander une exclusion mutuelle *ad-hoc* avec d'autre code stream. Plusieurs niveaux d'exclusion mutuelle sont disponibles :

  - Aucun code stream associé à la même queue de message ne peut s'exécuter en même temps,
  - Aucun code stream associé à la même instance du module ne peut s'exécuter en même temps,
  - Aucun code stream du même module stream ne peut s'exécuter en même temps (toutes instances confondues),

  - Aucun autre code stream ne peut se dérouler en même temps.

- L'environnement d'interruption. Les traitants d'interruption et d'événements asynchrones se déroulent dans cet environnement. De plus, on peut passer temporairement dans l'environnement d'interruption en masquant les interruptions. Cependant, l'opération inverse ne marche pas. Cet environnement ne permet ni les fautes de page, ni l'attente d'événements de synchronisation et il n'est pas recquérable.

Étant donné que notre système est assez complexe, nous avons besoin de plus de facultés possibles (fautes de page, attente sur les événements de synchronisation). Mais d'un autre côté nous devons aussi réagir à des événements qui nous placent dans un environnement défavorable :

- la routine qui nous avertit du déverrouillage d'un tampon entrée/sortie est invoquée dans l'environnement d'interruption,

- la routine qui nous avertit de l'arrivée d'un nouveau message stream en provenance d'un autre module est invoquée dans l'environnement stream.

Afin de résoudre cette contradiction, nous utilisons un serveur d'événements qui s'exécute en tant que *processus noyau*, donc dans l'environnement processus. Ce serveur reste en permanence bloqué sur une variable de condition. Le traitant d'interruption (et le traitant de message stream) place simplement un message décrivant l'événement sur cette file, et il signale la présence de ce message au serveur grâce à la variable de condition.

A.4.1 Gestion de la file de messages

Étant donné que le traitant d'interruption lui-même n'a pas le droit de se bloquer, nous devons faire en sorte que le rajout d'un message sur la file interne, et le retrait d'un message de cette file puissent se dérouler en concurrence sans avoir besoin de synchronisation explicite.

La manière habituelle d'attaquer ce genre de problème est d'assigner certains champs de la structure représentant la file à l'un des flots d'exécution concurrents, et les autres à l'autre. Chaque flot ne peut qu'écrire dans «ses» champs, mais il peut aussi lire les champs de l'autre flot.

Nous adoptons donc la structure suivante : L'ancre de la file d'attente comporte deux pointeurs : le pointeur de fin `DernierMessage` qui représente le message le plus récent, et le pointeur de début `PremierMessage` qui représente le message le plus vieux, qui sera traité en prochain. Le pointeur de fin est mis à jour par le traitant d'interruption (quand il place un message sur la file), tandis que le pointeur de début est mis à jour par le processus noyau quand il consomme un message. Les messages sont chaînés entre eux par un champ `Prochain`.

A.4.1.1 Procédure de mise en file d'un message

La figure A.6 décrit l'algorithme de mise en file d'un message.

```
Fonction EnfileMessage(File : TypeFile,
                       Message : TypeMessage) ;
   Prochain(DernierMessage(File)) = Message ;
   DernierMessage(File) = Message ;
   SignaleCondVar(File) ; /* le dormeur doit se réveiller */
```

FIG. A.6 – *Algorithme de mise en file d'un message*

A.4.1.2 Procédure de sortie de file d'un message

La procédure de sortie de file est un peu plus complexe. En effet, nous devons faire attention que la file ne soit jamais vide. En effet, comme la procédure de mise ne file doit mettre à jour le champ *Prochain* du dernier message de la file, il faut déjà que ce message existe. De plus, nous devons nous assurer que les messages sont traités dans un délai raisonnable, il est donc exclu d'attendre l'arrivée d'un nouveau message avant de traiter un message qui serait seul sur une file.

Nous résolvons ce problème en munissant chaque message d'un marqueur de validité. Initialement, un message est marqué comme valide. Après son traitement, un message est marqué comme invalide. Ceci nous permet de traiter les messages immédiatement, tout en les laissant sur la file pendant un certain temps, en attendant l'arrivée d'un nouveau message.

L'algorithme est décrit dans la figure A.7.

```
Fonction DéfileMessage(File : TypeFile) : TypeMessage ;
   DébutBoucle
      si Valide?(PremierMessage(File)) alors
         Traiter(PremierMessage(File)) ;
         MarqueCommeInvalide(PremierMessage(File)) ;
         ProchaineItération ;
      fin sin
      /* si nous passons ici, le premier message est invalide. */
      si PremierMessage(File) ≠ DernierMessage(File) alors
         /* nous pouvons éliminer un message invalide s'il n'est
          * plus le seul message de la file */
         Temporaire = PremierMessage(File) ;
         PremierMessage(File) = Prochain(Temporaire) ;
         Détruire(Temporaire) ;
         ProchaineItération ;
      fin si
      AttendreÉvénement(File) ;
   FinBoucle
```

FIG. A.7 – *Algorithme de sortie de file de message*

A.4.2 Variables de condition et exclusion mutuelle

Pour gérer nos files d'attente de messages nous utilisons des *variables de condition*. Nous utilisons aussi ce dispositif pour attendre les notifications de fin d'entrée-sortie disque. Les variables de conditions sont utilisées pour bloquer un flot d'exécution jusqu'à ce qu'une certaine condition devienne vraie. Ces variables mettent en jeu deux flots d'exécutions : le `Dormeur` et le `Réveilleur`. Le `Dormeur` attend que la condition devienne vraie, et le `Réveilleur` est invoqué par le flot qui rend la condition vraie.

A.4.2.1 Pourquoi pas des sémaphores ?

La figure A.8 représente une mise en œuvre simple du `Dormeur` et du `Réveilleur` à base de sémaphores.

```
Fonction Dormeur() ;
    P(Sémaphore) ;
```

```
Fonction Réveilleur() ;
    Lever(Soleil) ;
    V(Sémaphore) ;
```

FIG. A.8 – *Dormeur/Réveilleur avec sémaphores*

Mais malheureusement, cette mise en œuvre dépend de l'hypothèse implicite qu'il y a exactement un `Réveilleur` par `Dormeur` et vice-versa : il n'est par exemple pas possible d'invoquer le réveilleur « au cas où », car si jamais il n'y a pas de `Dormeur` attendant ce signal particulier, le sémaphore sera déphasé.

Par exemple, le Dormeur peut être invoqué par un allocateur de mémoire pour attendre que suffisamment de tampons mémoire se libèrent. De son côté le Réveilleur est invoqué par la routine de libération de tampons. Dans la plupart des cas, il n'y a pas pénurie, et donc pas de Dormeurs. Mais cependant, le Réveilleur est quand même invoqué pour toute libération de tampons. Ceci est nuisible, car cela « crédite » le sémaphore en temps de vaches pleines, et amène le Dormeur à ne pas attendre, même en période de vaches maigres, car le sémaphore porterait un « crédit » qui ne correspondrait pas à des tampons réellement disponibles.

Donc si on veut cette souplesse additionnelle, on doit envisager un autre mécanisme : les *variables de condition*. Une variable de condition ne peut pas porter un *crédit* comme un sémaphore, et le dormeur va dormir sur elle en tous les cas. Les variables de conditions s'utilisent toujours ensemble avec un test explicite de la condition attendue : si la condition est remplie, on ne bloque pas.

A.4.2.2 Utilisation sans précautions

D'habitude, les variables de condition s'utilisent conjointement avec les verrous d'exclusion mutuelle. Pour illustrer la raison pour ceci, nous allons montrer, grâce à la figure A.9 un scénario catastrophe qui risquerait de se passer si on n'utilisait pas de verrou :

Les deux procédures `Dormeur` et `Réveilleur` se déroulent dans des flots parallèles. Un ordonnanceur bascule entre ces deux flots « de manière aléatoire ». Nous

```
Fonction Dormeur() ;
   tant que SoleilLevé?() faire
      DormirSur(CondVar) ;
   FinBoucle
```

```
Fonction Réveilleur() ;
   Lever(Soleil) ;
   Réveiller(CondVar) ;
```

FIG. A.9 – *Dormeur/Réveilleur avec variable de condition seule*

supposons que la procédure `Dormeur` a juste franchi le test de la condition explicite, mais ne s'est pas encore endormi sur la variable de condition. À ce moment, par un hasard de l'ordonnancement, la procédure `Réveilleur` est exécutée entièrement : Elle a rendue la condition vraie, et réveillé la variable de condition. Cependant, à ce moment là, personne ne dormait sur la variable de condition, donc le signal s'est perdu. Quand le `Dormeur` avance, il se bloque sur la condition, et y reste éternellement, car le signal a déjà été délivré.

A.4.2.3 Solution classique

La solution classique à ce problème est d'utiliser un verrou d'exclusion mutuelle en plus de la variable de condition. La primitive `DormirSur` prend maintenant deux paramètres : la variable de condition, et un verrou. Le relâchement du verrou, et le blocage sur la variable de condition se font de manière atomique. Les procédures `Dormeur` et `Réveilleur` de la figure A.10 illustrent l'utilisation correcte des variables de condition : le verrou nous garantit que le scénario du premier exemple devient impossible : En effet, l'action de rendre la condition vraie et de réveiller le dormeur se déroule atomiquement dans une section critique en exclusion mutuelle de l'action de tester la condition et de s'endormir.

```
Fonction Dormeur() ;
   Verrouiller(Verrou) ;
   tant que SoleilLevé?() faire
      DormirSur(Verrou,CondVar) ;
   FinBoucle
   Déverouiller(Verrou) ;
```

```
Fonction Réveilleur() ;
   Verrouiller(Verrou) ;
   Lever(Soleil) ;
   Réveiller(CondVar) ;
   Déverrouiller(Verrou) ;
```

FIG. A.10 – *Dormeur/Réveilleur avec variable de condition et verrou*

Malheureusement, cette solution n'est pas adaptée à nos besoins, car elle nécessite le blocage éventuel du `Réveilleur` sur le verrou. Or dans notre contexte le `Réveilleur` se déroule dans un environnement d'exécution qui interdit le blocage (interruption ou traitant stream). Nous avons donc dû trouver une autre solution.

A.4.2.4 Notre solution

AIX nous permet de couper la primitive `DormirSur` en deux : `Sensibiliser` et `Dormir`. La primitive `Sensibiliser` rend le flot d'exécution sensible à tout signal de réveil sur la variable de condition `CondVar`. La primitive `Dormir` bloque le flot

d'exécution courant jusqu'à réception du signal décrit lors du dernier appel à `Sensibiliser`. Cependant, si le signal a *déjà* été reçu avant l'appel à `Dormir` (mais après l'appel à `Sensibilise`), l'appel ne bloque pas, mais continue tout de suite. Cela revient effectivement à autoriser un «crédit» sur la variable de condition. Cependant la situation ne comporte pas les inconvénients du sémaphore, étant donné que le «compte» n'est créditable que pendant un bref laps de temps au bout duquel se passe une opération qui vide le compte entièrement (la primitive `Dormir`).

D'après le manuel en ligne, les concepteurs d'AIX ont fourni ces primitives afin de permettre l'usage de plus d'un verrou (ou l'usage de verrous de types «exotiques») avec une variable de condition, permettant une mise en œuvre de `Défile` comme celle illustrée dans la figure A.11.

```
Fonction Dormeur() ;
   Verrouiller(Verrou1) ;
   Verrouiller(Verrou2) ;
   tant que SoleilLevé?()) faire
      Sensibiliser(CondVar) ;
      Déverouiller(Verrou2) ;
      Déverouiller(Verrou1) ;
      Dormir(CondVar) ;
      Verouiller(Verrou1) ;
      Verouiller(Verrou2) ;
   FinBoucle
   Déverouiller(Verrou2) ;
   Déverouiller(Verrou1) ;
```

FIG. A.11 – *Primitives de blocage sur variable de condition améliorées*

Cependant, ces primitives sont aussi utiles pour résoudre notre problème, comme l'illustre la figure A.12.

```
Fonction Dormeur() ;
   Sensibilise(CondVar) ;
   tant que SoleilLevé?() faire
      Dormir() ;
   FinBoucle
   Désensibilise() ;
```

```
Fonction Réveilleur() ;
   Lever(Soleil) ;
   Réveiller(CondVar) ;
```

FIG. A.12 – *Dormeur/Réveilleur correct avec variable de condition seule*

En effet, si le `Réveilleur` est exécuté entre le test de la condition et l'appel à `Dormir`, il aura «préréveillé» le `Dormeur` qui ne va donc pas s'endormir. La primitive `Désensibilise` permet de défaire l'association entre le flot d'exécution du `Dormeur` et de la variable de condition, dans le cas où la condition serait déjà remplie lors du test.

B

Interfaces

B.1 Externe

Cette section décrit l'interface externe offerte aux modules autres que ceux du stockage.

B.1.1 Interface externe du service générique de journalisation

Les messages décrits dans cette section, à l'exception de `LOGe_LogicalCreate` sont destinés au gestionnaire de journalisation local au site expéditeur de la requête. Ce gestionnaire les examine, et les relaye au site définitif s'ils concernent une zone qui réside sur un autre site.

B.1.1.1 Écriture d'un enregistrement dans le journal

Un message de type `LOGe_WjllWrite` déclenche l'écriture d'un enregistrement dans le journal. Il comporte dans son second en-tête les paramètres scalaires suivants :

id Ce paramètre identifie le journal logique auquel l'enregistrement appartient.

date Ce paramètre contient une estampille qui identifie l'instant où cet enregistrement a été généré. Il n'est pas utilisé par le service générique, mais est accessible au PJS lors de la reprise.

Handle Ce paramètre, contient un identificateur unique de l'enregistrement. Cet identificateur sera utilisé plus tard pour déclarer l'enregistrement comme obsolète

À la suite de l'en-tête à paramètres, le message contient le contenu de l'enregistrement. L'en-tête de l'enregistrement, tel qu'il figurera dans le journal, n'est pas inclus dans le message, mais sera construit à partir des données de l'en-tête de paramètres.

Une requête d'écriture est asynchrone, et ne donne donc pas lieu à une réponse.

B.1.1.2 Préparation d'un journal logique

Un message de type `LOGe_LogicalPrepare` contient un en-tête à paramètres de même format que celui d'un message d'écriture. Le contenu du message est un champ de bits identifiant les sites qui ont participé à la transaction.

Une requête de préparation est synchrone.

B.1.1.3 Validation d'un journal logique

Un message de type `LOGe_LogicalCommit` contient un en-tête à paramètres de même format que celui d'un message d'écriture. Il ne contient pas de partie données.

Une requête de validation est synchrone.

B.1.1.4 Abandon d'un journal logique

Un message de type `LOGe_LogicalAbort` contient un en-tête à paramètres de même format que celui d'un message d'écriture. Il ne contient pas de partie données.

Une requête d'abandon est synchrone.

B.1.1.5 Synchronisation explicite

Un message de type `LOGe_LogicalSync` contient un en-tête à paramètres de même format que celui d'un message d'écriture. Comme ce message ne donne pas lieu à l'écriture d'un nouvel enregistrement[20], les paramètres `date` et `Handle` ne sont pas utilisés. Seul l'identificateur de journal est donc utilisé. Un message de type synchronisation ne contient pas de partie données. Une requête de synchronisation explicite est synchrone.

B.1.1.6 Expiration d'enregistrement

Un message de type `LOGe_LogicalExpire` contient une liste d'identificateurs d'enregistrements à faire expirer. Il est asynchrone. Afin de permettre le routage, la zone désignée dans l'en-tête de routage doit être la même que celle désignée dans le message à faire expirer.

B.1.1.7 Création d'un journal logique

Un message de type `LOGe_LogicalCreate` est synchrone, et provoque l'allocation d'un identificateur de journal logique. Cette requête peut toujours être servie par le site local, car chaque instance du service générique de journalisation dispose d'une plage préallouée d'identificateurs de journal logique.

Ce message n'a qu'un seul paramètre (de retour) qui représente l'identificateur alloué.

B.1.2 Interface externe du gestionnaire de volumes

Le gestionnaire de volumes accepte les messages suivants :

20. Seul les enregistrements existants sont chassés sur disque.

B.1.2.1 Écriture d'une zone dans un volume

Un message de type `VOLe_Write` sert à écrire le contenu d'une zone dans son volume. L'étendue de la zone est décrite dans l'en-tête de routage. Ce message ne contient pas d'en-tête de paramètres scalaires. Après l'en-tête de routage suivent tout de suite le nouvel état de la zone. C'est un message synchrone.

B.1.2.2 Lecture d'une zone à partir d'un volume

Un message de type `VOLe_Read` sert à lire le contenu d'une zone à partir d'un volume. L'étendue de la zone est décrite dans l'en-tête de routage. La réponse à ce message contient le contenu de la zone. Si le segment contenant la zone a été bloqué par un message `VOLe_Lock`, la réponse sera retardée jusqu'à la réception d'un message `VOLe_Unlock`.

B.1.2.3 Destruction d'un segment

Un message de type `VOLe_Delete` provoque la destruction du segment qui contient la zone décrite dans son en-tête de routage. Il n'y a pas d'interface de création, en effet la création se fait de manière implicite lors de la première tentative d'accès à un segment.

B.2 Interne

Cette section décrit l'interface entre les différentes parties du service de stockage.

B.2.1 Interface du gestionnaire de volume offerte au SGJ

B.2.1.1 Blocage d'un segment

Un message de type `VOLe_Lock` bloque le segment contenant la zone décrite dans son en-tête de routage. Tout accès en lecture à un segment bloqué est mis en attente jusqu'au déblocage du segment. Ce type de requête est utilisé lors de la reprise après panne pour rendre temporairement inaccessible les segments qui n'ont pas été récupérés à cause d'un journal se trouvant dans un état indéterminé. Il n'est pas nécessaire de bloquer les écritures, étant donné que dans Arias, un segment doit de toute manière d'abord être lu en mémoire d'exécution afin de pouvoir être modifié.

B.2.1.2 Déblocage d'un segment

Un message de type `VOLe_Unlock` débloque le segment contenant la zone décrite dans son en-tête de routage. Ce message déclenche le relancement des requêtes de lecture mis en attente sur ce segment.

B.2.2 Interface du gestionnaire de volume offerte au PJS

B.2.2.1 Lecture d'une zone à partir d'un volume, pour récupération

Un message de type `VOLe_PrivRead` permet de lire une zone, tout comme le message `VOLe_Read`. Cependant, il n'est pas affecté par l'état bloqué ou non du segment. Ce type de message est nécessaire avant de permettre au PJS d'avoir accès au segment qu'il est chargé de récupérer. Afin de ne pas créer des dépendances entre segments, ce qui fausserait les prémisses de la section 5.7.3, le protocole de récupération doit s'abstenir d'utiliser ce type de message pour accéder à des segments autres que ceux dont il est responsable de la récupération.

B.3 Messages circulant entre les instances de SGJ des différents sites

B.3.1 Messages de journalisation

Les messages décrits dans la section B.1.1 (à l'exception de `LOGe_LogicalCreate`) sont d'abord adressés par le PJS à l'instance locale du SGJ. Celui-ci change le type de ces messages de `LOGe_Logical???`[21] en le type `LOGe_Wjll???` équivalent. Le nouveau message est ensuite redirigé vers le site définitif de traitement.

Les messages de type `LOGe_Wjll???` ont la même syntaxe de paramètres que les messages de type `LOGe_Logical???`.

B.3.2 Protocole d'accord pour la récupération

Cette sous-section décrit les messages qui circulent entre les différentes instances de SGJ, et qui servent à mettre ces instances d'accord au sujet de l'état des journaux logiques répartis.

B.3.2.1 État des journaux locaux

Le message `RecoveryLogstate` communique l'état d'un certain nombre de journaux logiques locaux à récupérer au site destinataire du message. Il contient une suite de structures contenant les champs suivants :

state désigne l'état du journal (Validé, Abandonné ou Indéfini),

kwown est un champ de bits énumérant les sites qui sont d'accord avec l'expéditeur sur l'état du journal,

id est l'identificateur du journal décrit.

21. Dans cette notation, les trois points d'interrogation représentent l'un des types de messages particuliers : `Write, Commit, Prepare, Abort, Sync, Expire`.

B.3.2.2 Signalisation de présence

Le but premier du message `RecoveryHello` est de signaler la présence de l'expéditeur aux autres sites. En réponse, les autres sites se font connaître par un message `RecoveryLogstate`. De plus le message `RecoveryHello` peut aussi comporter une liste d'états de journaux logiques locaux dans le même format que celle du message `RecoveryLogstate`.

Table des figures

2.1 État des pages avant chargement de la page X 16
2.2 État des pages après le chargement de la page X 16
2.3 Accès à la mémoire virtuelle 17
2.4 Un assemblage de blocs de journalisation dans Kitlog 24
2.5 Copie logique d'un journal 24
2.6 Journaux hiérarchisés 25
2.7 Structure d'un segment en Clio 25
2.8 Chaînage des enregistrements de compensation dans Aries 27
2.9 Situation relative des différents LSN 27
2.10 Reprise après panne dans Aries 28
2.11 Table d'implantation dans LFS 31
2.12 Architecture fonctionnelle de GEODE 32
2.13 Mise en œuvre de la journalisation et de l'atomicité 35
2.14 Architecture de Camelot 36
2.15 Architecture de Cricket sur un site 40
2.16 Intégration stockage et mémoire virtuelle 42

3.1 Support mémoire 45
3.2 Architecture générale du sous-système de cohérence et synchronisation 53
3.3 Les différents éléments du service de stockage 54
3.4 En-tête d'un enregistrement de journal 57
3.5 Les modules logiciels 59
3.6 Implantation des services à base de «stream» 62

4.1 Architecture du système 69
4.2 Répartition de journaux 71
4.3 États des journaux logiques après une panne 73
4.4 Diagramme de transitions d'états d'une page 77
4.5 Purge dy journal et début du journal 79

5.1 Architecture interne du service de stockage 81
5.2 Localisation de la fin du journal avec et sans retour 84
5.3 Algorithme de localisation de la fin de la partition physique 85
5.4 Algorithme de choix du bon ancre 86
5.5 Algorithme d'écriture de données 87
5.6 Métadonnées du journal physique 88
5.7 Algorithme d'une écriture d'un enregistrement 89

5.8 Lecture de l'enregistrement suivant, en tenant compte des cassures . . 90
5.9 Lecture de l'enregistrement précédent, en tenant compte des cassures 91
5.10 Algorithme d'initialisation de pointeurs de fin 92
5.11 Base de comparaison pour les positions des enregistrements 93
5.12 Enlèvement d'un élément du tas 94
5.13 Enlèvement d'un élément du tas avec nœud unaire 94
5.14 Enlèvement d'une feuille du tas 95
5.15 Insertion d'un élément dans un tas 95
5.16 Algorithme d'insertion d'un nœud dans un tas 96
5.17 Mise à jour des indicateurs d'équilibre entre la racine et le nœud enlevé 97

6.1 Influence du nombre des enregistrements 106
6.2 Influence de la taille des enregistrements 107
6.3 Influence du groupement des données 108
6.4 Ordonnancement de l'écriture des journaux 109
6.5 Influence du parallélisme des transactions 110

A.1 Pile de *stream* 123
A.2 Utilisation de multiplexeurs 125
A.3 Structure des messages *stream* 127
A.4 Pile stream de Arias 128
A.5 Exemple d'un message stream de Arias 130
A.6 Algorithme de mise en file d'un message 132
A.7 Algorithme de sortie de file de message 132
A.8 Dormeur/Réveilleur avec sémaphores 133
A.9 Dormeur/Réveilleur avec variable de condition seule 134
A.10 Dormeur/Réveilleur avec variable de condition et verrou 134
A.11 Primitives de blocage sur variable de condition améliorées 135
A.12 Dormeur/Réveilleur correct avec variable de condition seule 135

Table des matières

Chapitre 1
Introduction

1.1 Cadre de travail 6
1.2 Démarche 7
1.2.1 Objectifs et contraintes 7
1.2.2 Méthode proposée 7
1.2.3 Transactions dans une mémoire virtuelle répartie 7
1.3 Plan de la suite 8

Chapitre 2
Stockage des données : État de l'art

2.1 Interface des systèmes de stockage 11
2.1.1 Exemples 12
2.1.1.1 Monads 12
2.1.1.2 Clouds 13
2.1.1.3 Opal 15
2.1.1.4 Texas 16
2.1.2 Synthèse 17
2.2 Résistance aux pannes 18
2.2.1 Sauvegarde et reprise 18
2.2.1.1 Monads 19
2.2.1.2 Clouds 19
2.2.2 La journalisation 20
2.2.2.1 Problèmes et terminologie 20
2.2.2.2 Camelot 22
2.2.2.3 Kitlog 23
2.2.2.4 Clio 23

2.2.2.5 Aries 25
2.2.3 Systèmes de fichiers à base de journal (Log File Systems) . . . 29
2.2.3.1 Sprite LFS 29
2.2.3.2 Texas 30
2.2.3.3 GEODE 31
2.2.4 Versions ombres (System R) 33
2.2.5 Synthèse 33
2.3 Intégration de la mémoire virtuelle et du stockage 34
2.3.1 Camelot 35
2.3.2 Exodus 36
2.3.3 GEODE 37
2.3.3.1 Modèle d'exécution 38
2.3.3.2 Accès aux objets 38
2.3.4 Gestion de la répartition 38
2.3.5 Shore 39
2.3.6 Cricket 39
2.3.7 Synthèse 41
2.4 Conclusion 41

Chapitre 3
Présentation générale de Arias

3.1 Objectifs 43
3.2 Présentation générale de l'architecture 44
3.2.1 Service de partage des données 45
3.2.1.1 Gestion de la mémoire partagée 45
3.2.1.2 Gestion de la cohérence 46
3.2.2 Service de permanence 47
3.2.3 Service de protection 47
3.3 La gestion des segments 48
3.3.1 Manipulation des segments 48
3.3.1.1 Création 48
3.3.1.2 Destruction 49
3.3.1.3 Accès 49
3.3.2 Localisation 49
3.3.3 Allocation des segments 50

3.4 La cohérence et la synchronisation 51
3.4.1 L'approche Arias 52
3.4.2 Architecture générale du service de CeS 52
3.5 La permanence 54
3.5.1 La mémoire de stockage 55
3.5.2 Utilisation de l'espace de stockage par les paginateurs 55
3.5.3 Le journal 56
3.5.3.1 Modèle centralisé 58
3.5.3.2 Modèle réparti 58
3.6 Architecture des services d'Arias 59
3.6.1 Architecture logicielle 59
3.6.2 Mise en œuvre sur un système Unix 61
3.7 Validation 62
3.8 Comparaison avec l'existant 64
3.8.1 Gestion d'un espace virtuel unique 64
3.8.2 Cohérence et synchronisation 64
3.8.3 Permanence 64
3.8.4 Protection 64

Chapitre 4
Conception du système de stockage des données

4.1 Buts du service de stockage 67
4.1.1 Les propriétés *ACID* 67
4.1.2 Répartition 68
4.1.3 Souplesse 68
4.1.4 Performance et Intégration 69
4.2 Intégration du service de stockage dans son contexte général 69
4.2.1 Souplesse et format du journal 70
4.3 Récupération après une panne 70
4.3.1 Transactions multi-sites et validation à deux phases 71
4.3.2 Journalisation avant 72
4.3.3 Esquisse de l'algorithme de récupération 73
4.3.4 Souplesse et modularité 74
4.3.5 Gestion des enregistrements qui sont dans un état non défini . 74
4.4 Interaction avec la mémoire virtuelle 75

4.4.1 Mise en œuvre simple, qui utilise des pages entières 75
4.4.2 Mises à jour de l'image en différé 75
4.4.3 Définition de l'état cohérent 75
4.4.4 Granularité 76
4.4.5 Réutilisation d'une zone cohérente 76
4.4.6 Algorithme de vidage de pages 77
4.4.7 Diagramme de transition d'états 77
4.5 Purge d'enregistrements obsolètes 78
4.5.1 Contexte général 78
4.5.2 Le déroulement de la purge du journal en fonctionnement normal 78
4.5.2.1 Le rôle du PJS 78
4.5.2.2 Le rôle du paginateur 79
4.5.2.3 Le rôle du SGJ 79
4.5.2.4 Nature des identificateurs 79
4.6 Disponibilité et durabilité 79
4.6.1 Réplication 79

Chapitre 5
Mise en œuvre du service de stokage

5.1 Architecture interne du sous-système de stockage 81
5.2 Le gestionnaire de volume 82
5.2.1 Atomicité des mises à jour 82
5.3 Le journal physique 83
5.3.1 Étendue du journal 83
5.3.2 Localisation de la fin du journal 83
5.3.2.1 Les problèmes et les solutions simplifiées 83
5.3.2.2 Notre solution 84
5.3.3 Le début du journal 84
5.3.4 Sécurité de l'ancre du journal 85
5.3.5 Sécurité des écritures et bande passante 85
5.3.6 Organisation du journal physique 86
5.3.7 L'indicateur de cassure 87
5.3.7.1 Cassures lors d'une panne 88
5.3.7.2 Cassures lors des synchronisations 88

5.3.8 Algorithmes . . . 89
5.3.8.1 Écriture . . . 89
5.3.8.2 Lecture en avant . . . 89
5.3.8.3 Lecture en arrière . . . 90
5.3.8.4 Reprise après panne . . . 91
5.4 Multiplexage des JLL sur un JP . . . 91
5.5 Éclatement des JL en JLL . . . 92
5.6 Gestion de l'obsolescence . . . 93
5.6.1 Enlèvement d'un élément du tas . . . 93
5.6.2 Insertion d'un nouvel élément dans un tas . . . 94
5.6.3 Maintien des indicateurs d'équilibre . . . 95
5.7 Récupération après une panne . . . 96
5.7.1 Objectifs et modèle de récupération après une panne . . . 96
5.7.2 Protocole d'accord réparti . . . 97
5.7.3 Mise en attente de journaux logiques et de segments . . . 97
5.7.4 Les déroulement de l'algorithme de récupération . . . 98
5.7.5 Les structures de données utilisées . . . 99
5.7.6 La gestion des journaux et segments bloqués . . . 99
5.7.7 Le protocole d'accord entre sites . . . 100
5.7.7.1 Le format des messages . . . 100
5.7.7.2 Traitement d'un message . . . 101

Chapitre 6
Évaluation du système

6.1 Environnement d'essai . . . 103
6.2 Quantités mesurées . . . 103
6.3 Paramètres explorés . . . 104
6.4 Résultats et interprétation . . . 105
6.4.1 Latence de validation d'une transaction courte . . . 105
6.4.2 Influence du nombre d'enregistrements journalisés . . . 105
6.4.3 Influence de la taille des enregistrements . . . 105
6.4.4 Influence du groupement des données . . . 107
6.4.5 Comportement d'une pile streams allégée . . . 108
6.4.6 Influence du parallélisme des transactions . . . 109

6.5 Conclusion 110

Chapitre 7
Conclusion

7.1 Aspects fonctionnels 113
7.2 Mise en œuvre 114
7.3 Contributions 115
7.4 Évaluation 115
7.5 Perspectives 116

Bibliographie **117**

Annexes

Annexe A
Interface entre Arias et AIX

A.1 Communication par stream 123
A.1.1 Les briques de base d'un stream 124
A.1.1.1 Tête de stream 124
A.1.1.2 Le pilote de stream 124
A.1.1.3 Les modules stream 124
A.1.2 Les messages et les files 124
A.1.2.1 Messages 124
A.1.2.2 Files d'attente 124
A.1.3 Les multiplexeurs 125
A.1.4 Traitement d'un message par un pilote ou par un module . . . 125
A.1.5 Structure d'un message stream 126
A.1.5.1 Mise à jour de messages 126
A.1.5.2 Partage de parties arbitraires d'un message 126
A.1.5.3 Types associés au blocs de messages 127
A.2 La pile de stream Arias 127
A.3 Formats de messages utilisés dans Arias 129
A.3.1 Formats des messages pour le service de stockage 129
A.4 Environnements d'exécution 130
A.4.1 Gestion de la file de messages 131

A.4.1.1 Procédure de mise en file d'un message 131
A.4.1.2 Procédure de sortie de file d'un message 132
A.4.2 Variables de condition et exclusion mutuelle 133
A.4.2.1 Pourquoi pas des sémaphores? 133
A.4.2.2 Utilisation sans précautions 133
A.4.2.3 Solution classique 134
A.4.2.4 Notre solution 134

Annexe B
Interfaces

B.1 Externe 137
B.1.1 Interface externe du service générique de journalisation 137
B.1.1.1 Écriture d'un enregistrement dans le journal 137
B.1.1.2 Préparation d'un journal logique 138
B.1.1.3 Validation d'un journal logique 138
B.1.1.4 Abandon d'un journal logique 138
B.1.1.5 Synchronisation explicite 138
B.1.1.6 Expiration d'enregistrement 138
B.1.1.7 Création d'un journal logique 138
B.1.2 Interface externe du gestionnaire de volumes 138
B.1.2.1 Écriture d'une zone dans un volume 139
B.1.2.2 Lecture d'une zone à partir d'un volume 139
B.1.2.3 Destruction d'un segment 139
B.2 Interne 139
B.2.1 Interface du gestionnaire de volume offerte au SGJ 139
B.2.1.1 Blocage d'un segment 139
B.2.1.2 Déblocage d'un segment 139
B.2.2 Interface du gestionnaire de volume offerte au PJS 140
B.2.2.1 Lecture d'une zone à partir d'un volume, pour récupération 140
B.3 Messages circulant entre les instances de SGJ des différents sites . . . 140
B.3.1 Messages de journalisation 140
B.3.2 Protocole d'accord pour la récupération 140
B.3.2.1 État des journaux locaux 140
B.3.2.2 Signalisation de présence 141

Table des figures **143**

Design and implementation of a reliable and extensible storage server for a distributed persistent object oriented system

In this thesis, we describe the design and implementation of a reliable and extensible storage server. This work has been done in the framework of Sirac, a distributed system supporting persistent objects. The goal of Sirac is to supply services for the support of persistent distributed objects and for the construction of distributed applications.

The key ideas that have oriented this study are the flexibility of the offered services and the cooperation among the subsystems. Flexibility, as made possible by the modular design, enhances performances, as applications only must pay the price of the services that they use. Cooperation (for example between the storage subsystem and the pager) allows the various modules to make informed decisions.

In this thesis, we start out by presenting the state of the art in three directions. We first study the way how a big storage space can be presented to the applications. In the second part, we analyze the various implementations of reliable storage, focusing on atomicity. The third part shows how the newer systems achieve to reconcile both areas.

In the third chapter, we present briefly the overall Arias system, with its various subsystems : security, consistency, synchronization and storage. Within the services, we distinguish on one hand generic low level modules, which implement *mechanism* and on the other hand application specific high level modules, which define *policy*. Some subsystems are present in every Arias system, such as the consistency and the synchronization services, whereas others, such as security and storage services, are optional.

In the fourth and fifth chapter, we zoom in on the storage service. The generic storage server is divided in two : first the volume manager, whose purpose is to ensure the long term durability of the data, and second the generic logging service, which ensures the atomicity of the transactions.

Our system has been implemented on top of AIX, and the communication among the various modules relies on the *streams* mechanism.

The performances of our system are good, and approach the limits allowed by the hardware in favorable cases. Our future projects include the implementation of a vast range of specific logging protocols, the support for replicated volumes, and optimization of the volume manager.

Keywords: fault tolerance, distributed shared memory, log, flexibility, modularity

Conception et réalisation d'un service de stockage fiable et extensible pour un système réparti à objets persistants

Cette thèse décrit la conception et la mise en œuvre d'un service de stockage fiable et extensible. Les travaux ont été faits dans le cadre de Sirac, un système réparti à objets persistants. L'objectif de Sirac est de fournir des services pour le support d'objets persistants répartis et pour la construction d'applications réparties.

Les deux idées qui ont dirigé cette étude sont la souplesse des services offerts et la coopération entre les sous-systèmes. La souplesse, rendue possible par la conception modulaire du système, améliore les performances, étant donné que les applications doivent seulement payer le prix des services qu'elles utilisent. La coopération (par exemple entre le stockage et la pagination) permet aux différents modules de prendre des décisions en connaissance de cause.

La thèse présente dans le second chapitre un état de l'art en trois parties. La première partie s'attache à étudier la manière dont un grand espace de stockage unique peut être présenté aux applications. La deuxième partie analyse la mise en œuvre du stockage fiable en étudiant notamment différentes réalisations de l'atomicité. La troisième partie enfin montre comment ces deux aspects sont mariés dans les systèmes modernes.

Dans le troisième chapitre, nous faisons un rapide tour d'horizon d'Arias et de ses différents sous-systèmes : protection, cohérence, synchronisation et stockage. Au sein des différents services, nous distinguons d'un côté des modules génériques de bas niveau, et d'un autre côté des modules spécifiques aux applications. Les modules génériques mettent en œuvre les *mécanismes* tandis que les modules spécifiques définissent la *politique*. Certains sous-systèmes sont toujours présents, comme la gestion de la cohérence et de la synchronisation, alors que d'autres, comme par exemple la gestion de la protection ou la gestion de la permanence, sont optionnels.

Dans les quatrième et cinquième chapitres, nous nous concentrons sur le service de stockage. Le service générique de stockage est subdivisé en deux parties : d'abord un gestionnaire de volume, qui assure la pérennité des données, et puis un service de journalisation, qui assure l'atomicité des transactions.

Ce système a été mis en œuvre au dessus d'AIX, et la coopération entre les différents modules s'appuie sur le mécanisme des *streams*.

Les performances de notre système sont bonnes, et s'approchent des limites imposées par le matériel dans les cas favorables. Les projets futurs incluent la fourniture d'un vaste éventail de protocoles de journalisation spécifiques, le support de volumes dupliqués ainsi que l'optimisation du gestionnaire du volume.

Mots-clés: tolérance aux pannes, mémoire virtuelle répartie, journal, souplesse, modularité