Chapitre 5

Implantation des Espaces de Travail

I. Introduction 82
II. Objectifs de Conception 83
III. Accès aux Pages Persistantes 88
IV. Structure MultiThreadée des Espaces de Travail 105
V. Accès aux Objets 119
VI. L'interface Langage: le typage des objets. 128
VII. Versions Alternatives et Travail Coopératif 139
VIII. Conclusion 146

I. Introduction

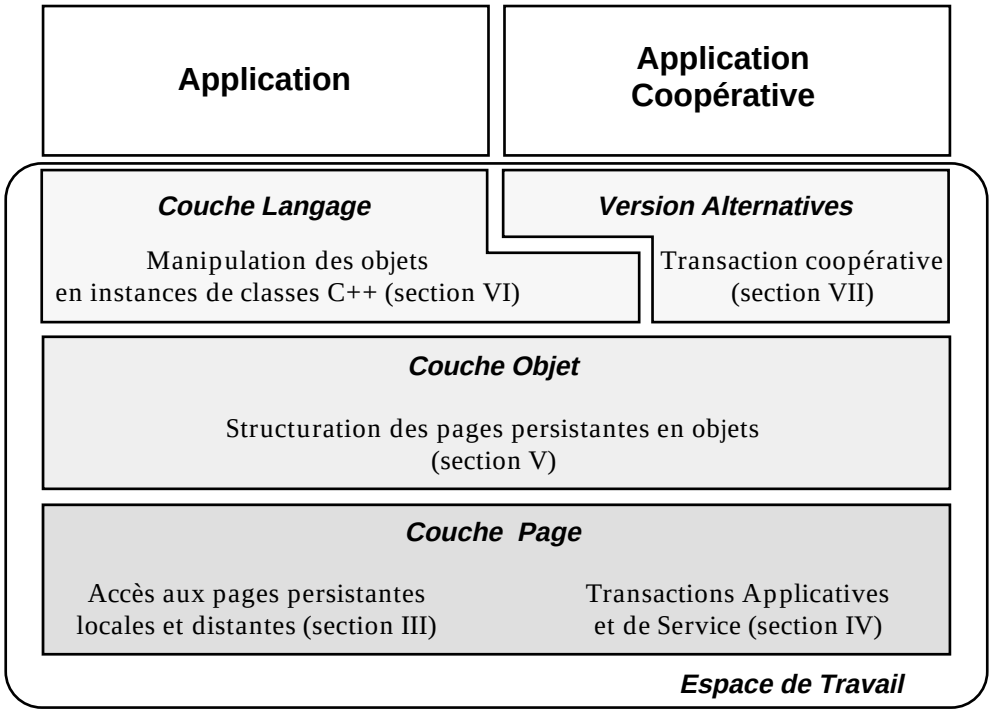
L'implantation des Espaces de Travail est basée sur l'utilisation des mécanismes récents des systèmes d'exploitation, comme le Memory-Mapping et le MultiThreading. Cette approche nous a conduit à choisir la page comme granularité de communication, de réplication et de verrouillage. La section II rappelle les propriétés de ces mécanismes et montre l'intérêt de ceux-ci pour l'accès aux données persistantes et dans la conception de communications asynchrones.

Nous présenterons successivement les différentes couches qui constituent l'Espace de Travail. La section III présentera la structure de la couche d'accès aux pages persistantes. Ces accès utilisent les nombreuses facettes du *Memory-Mapping* afin de rendre la gestion de la persistance transparente au code des transactions.

La section IV présentera la structure multi-threadée des Espaces de Travail. Cette structure permet aussi bien d'exécuter en parallèle les transactions applicatives que les transactions attachées aux abonnements des clients. Nous verrons que cette structure assure aux Espaces de Travail un traitement asynchrone des services présentés dans les chapitres 3 et 4.

La section V présentera la couche Objet qui structure le contenu des pages persistantes en objets courts ou longs. Dans la section VI, l'interface Langage manipule les instances des classes C++ définies par l'utilisateur.

Enfin, nous terminerons par la section VII qui présentera un des mécanismes principaux du travail coopératif : celui des versions alternatives. Une version alternative est un objet composite qui ne contient que la valeur des objets modifiés depuis la version alternative précédente.



II. Objectifs de Conception

L'Espace de Travail est un noyau d'un Gérant d'Objets Persistants destiné à remplir les fonctions de serveur comme celui de client. Sa conception générique s'efforce de tenir compte des exigences des deux.

Nos objectifs de conception des Espaces de Travail sont les suivants :

- Approcher les performances des applications non persistantes,
- Concevoir l'Espace de Travail de façon générique pour réaliser les fonctions du client et celles du serveur,
- Intégrer l'Espace de Travail dans l'environnement existant.

II.1. Approcher les Performances des Applications non persistantes

L'utilisation des Gérants d'Objets Persistants reste encore marginale dans le développement d'applications : malgré un certain nombre d'inconvénients, les systèmes de fichiers demeurent le support de persistance utilisé pour archiver les données d'une application. Une application, qui utilise des fichiers pour archiver ces données, installe en mémoire l'ensemble des objets de sa "base" avant d'effectuer ses calculs. L'application se termine en sauvegardant l'ensemble des objets modifiés ou non dans le fichier. Les inconvénients de cette approche sont multiples [Weiner94] :

- les formats disque (souvent un format texte) et mémoire des objets sont fréquemment différents; ces derniers sont convertis avant d'être chargés dans la mémoire de l'application et avant d'être sauvés sur disque.
- la persistance des associations entre objets est complexe à réaliser, car les pointeurs ne sont valides que durant l'exécution d'un processus.
- l'ensemble des données est chargé en mémoire en début de programme. Quand la base est importante, l'installation pénalise les applications qui n'accèdent qu'à une partie des objets.
- Le fichier est un grain trop gros de partage entre les applications concurrentes.

Les Gérants d'Objets apportent des solutions à chacun de ces problèmes. Cependant, un des principaux freins à la généralisation de ceux-ci est la performance moyenne qu'offrent ces derniers lors des accès aux objets. Ces mauvaises performances sont dues en général aux mécanismes d'accès aux objets. Chaque objet est accessible via un identifiant invariant qui doit être traduit vers l'adresse mémoire contenant l'objet. Cette traduction, qui est répétée lors chaque accès à un objet, nécessite souvent plusieurs dizaines d'instructions. Ce surcoût est souvent intolérable pour une application telle qu'un simulateur électrique de d'un atelier de CAO-VLSI qui référence plusieurs milliers de fois le même objet au cours d'une simulation.

Plusieurs Gérants d'Objets [Lamb91, Gruber92, Singhal92] optimisent l'opération de déréférenciation en profitant de la MMU des processeurs ou des mécanismes systèmes qui en permettent l'accès : le coût d'une déréférenciation est alors celui d'un accès par pointeur d'un langage de programmation non persistant, une fois que l'objet a été installé dans la mémoire de l'application. L'utilisation de tels mécanismes contraint néanmoins le Gérant d'Objets à manipuler des pages plutôt que des objets.

Notre conception de l'Espace de Travail s'oriente vers de tels mécanismes pour ne pas pénaliser l'application par les déréférenciations d'identifiants d'objets. Ce choix implique que la structure de l'Espace de Travail est orientée vers la manipulation des pages persistantes lors des échanges entre Espaces de Travail ou pour le contrôle de concurrence dans les espaces de données entre transactions.

II.2. Conception Générique de l'Espace de Travail

En général, la structure d'un Gérant d'Objets distingue deux composants : le client et le serveur. Le client est une mémoire contenant des objets ou des pages sur lesquels l'application effectue les consultations et les modifications. Le client est muni d'un mécanisme de "swap" permettant à l'application d'utiliser plus d'objets que la mémoire physique du client ne peut en stocker. Le serveur possède les fonctions de contrôle de concurrence et de reprise sur panne. Le serveur maintient un buffer des pages demandées par les clients qui lui évite d'aller les rechercher sur l'archive (le disque) lors des demandes suivantes.

La structure de WEA ne distingue pas le client du serveur : il n'existe qu'un seul composant générique, l'Espace de Travail. L'Espace de Travail utilise l'espace de données comme un ensemble de pages d'objets provenant d'une archive locale ou d'une archive distante (les pages sont importées depuis un serveur). L'espace de données permet à plusieurs transactions simultanées d'accéder aux pages d'objets, sans se soucier de leur localisation (archive locale ou distante). Les transactions exécutent des applications ou des services. Le contenu du service peut exporter des pages d'objets (services de Données et Coopératifs) ou bien réaliser un calcul d'une application consultant et/ou modifiant plusieurs pages de l'espace de données (services d'Opérations et Mixtes).

L'Espace de Travail utilise des mécanismes génériques communs au client ou serveur. Toutefois dans le cadre d'une utilisation particulière, des optimisations peuvent améliorer localement les performances : si l'Espace de Travail n'exécute qu'une seule application (une seule transaction à la fois), la déréférenciation est améliorée par une technique de "swizzling" présentée dans la section VI.2.3. Côté serveur, les transactions du service de Données n'ont pas besoin d'espace de modifications privées; les threads serveurs partagent donc les mêmes mappings de la base (section IV.2.3).

II.3. Intégrer l'Espace de Travail dans un Environnement Système

Les technologies tant au niveau matériel qu'au niveau système d'exploitation ont beaucoup évolué. Les architectures multiprocesseurs se généralisent et offrent la puissance du parallélisme sur de simples stations de travail. En corrélation, de nouveaux systèmes d'exploitation ont suivi

cette évolution du matériel et permettent notamment d'exploiter le parallélisme [Kleiman91]. Ces systèmes proposent de masquer le multiprocesseur par les notions de tâches et d'activités.

Les systèmes d'exploitation récents proposent de réaliser les accès à un fichier au travers d'accès mémoire dans l'espace d'adressage de l'application. Cette technique appelée "Memory-Mapping" évite d'utiliser des appels systèmes pour accéder aux pages du fichier. La gestion de la mémoire de la machine est optimale car il n'existe plus de différence entre les pages d'un processus et celles cachant des morceaux des fichiers disques.

Nous avons choisi pour implanter l'Espace de Travail d'utiliser ces concepts introduits récemment dans les systèmes d'exploitation. Le Memory-Mapping est utilisé pour constituer l'espace de données de l'Espace de Travail par couplage des fichiers d'archives locales ou des fichiers d'images distantes. Les mécanismes associés de protection de la mémoire nous permettent de détecter implicitement les demandes de verrous par les transactions. Le MultiThreading est utilisé pour implanter les différentes activités d'un Espace de Travail. Les deux sections suivantes rappellent les fonctionnalités de ces deux mécanismes. La troisième introduit l'utilisation de ces mécanismes dans la conception de l'Espace de Travail.

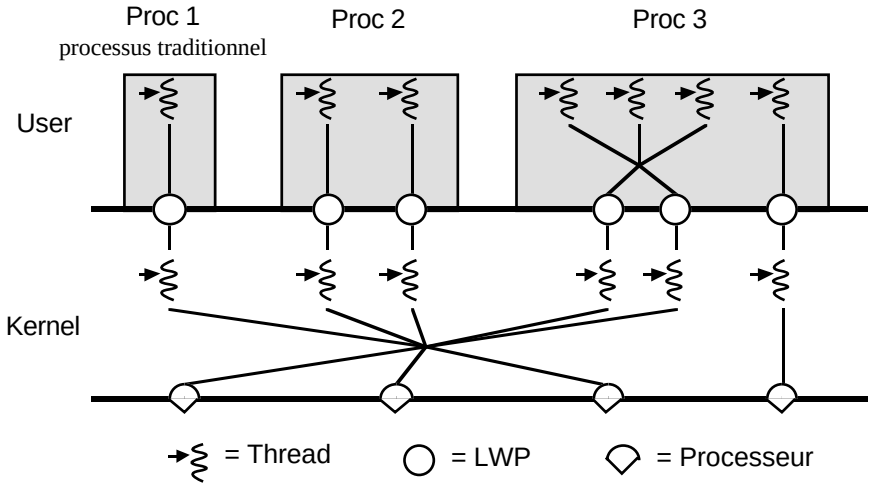
II.3.1. Le MultiTheading

L'abstraction du processus UNIX n'est plus adaptée aux exigences des applications modernes, en particulier, celles utilisant des multiprocesseurs à mémoire partagée. Dans les systèmes récents comme Solaris, Mach et Chorus, la notion de processus a été remplacée par les notions de tâche et d'activité. La tâche correspond à un espace commun de ressources systèmes incluant des ports de communications et un espace d'adressage. Cette tâche est partagée par plusieurs activités qui sont des unités d'utilisation de la CPU. Le contexte d'une activité se résume alors à un compteur d'instructions et d'une pile privée d'exécution; l'activité a donc un coût faible de gestion par rapport au processus.

- Sur Mach [Baron87], la tâche est une Task et l'activité la Thread.
- Sur Chorus [Herrmann88], la tâche est un Actor et l'activité la Thread.
- Sur Solaris [SunSoft93a], la tâche conserve le nom de Process et l'activité celui des Light Weighted Process (LWP). Il ne faut pas confondre les LWPs de Solaris avec les lwp's qui étaient dans une bibliothèque de SunOS4.x [Kepecs85]: cette bibliothèque ne permet pas d'exploiter le parallélisme et souffre de ne pas permettre plusieurs appels au système comme le propose Solaris qui est conçu comme un système ré-entrant.

Ces systèmes proposent également une notion "plus légère" de l'activité, la sous-activité. La sous-activité est une notion interne à la tâche; elle n'est pas connue du noyau du système : les sous-activités sont exécutées par une activité (ou un groupe d'activités) en temps partagé sans intervention du noyau. L'application peut ainsi instancier un très grand nombre de sous-activités à faible coût, et adapter son exécution au parallélisme qui lui est réellement nécessaire [Powell91, Stein92]. La sous-activité est appelée C-Thread [Cooper88] sur Mach et Thread sur Solaris (figure 5.1). Enfin, notons que la plupart des bibliothèques de MultiThreading suivent les recommandations POSIX [IEEE92]. Dans la suite du texte, nous utiliserons le terme de Thread pour désigner conjointement les activités et les sous-activités et pour éviter de les confondre avec les activités définie dans le modèle des Espaces de Travail (chapitre 3, section III.2.5).

Figure 5.1 : Structure multi-threadée de Solaris 2 (d'après [Powell91]).



II.3.2. Le Memory-Mapping

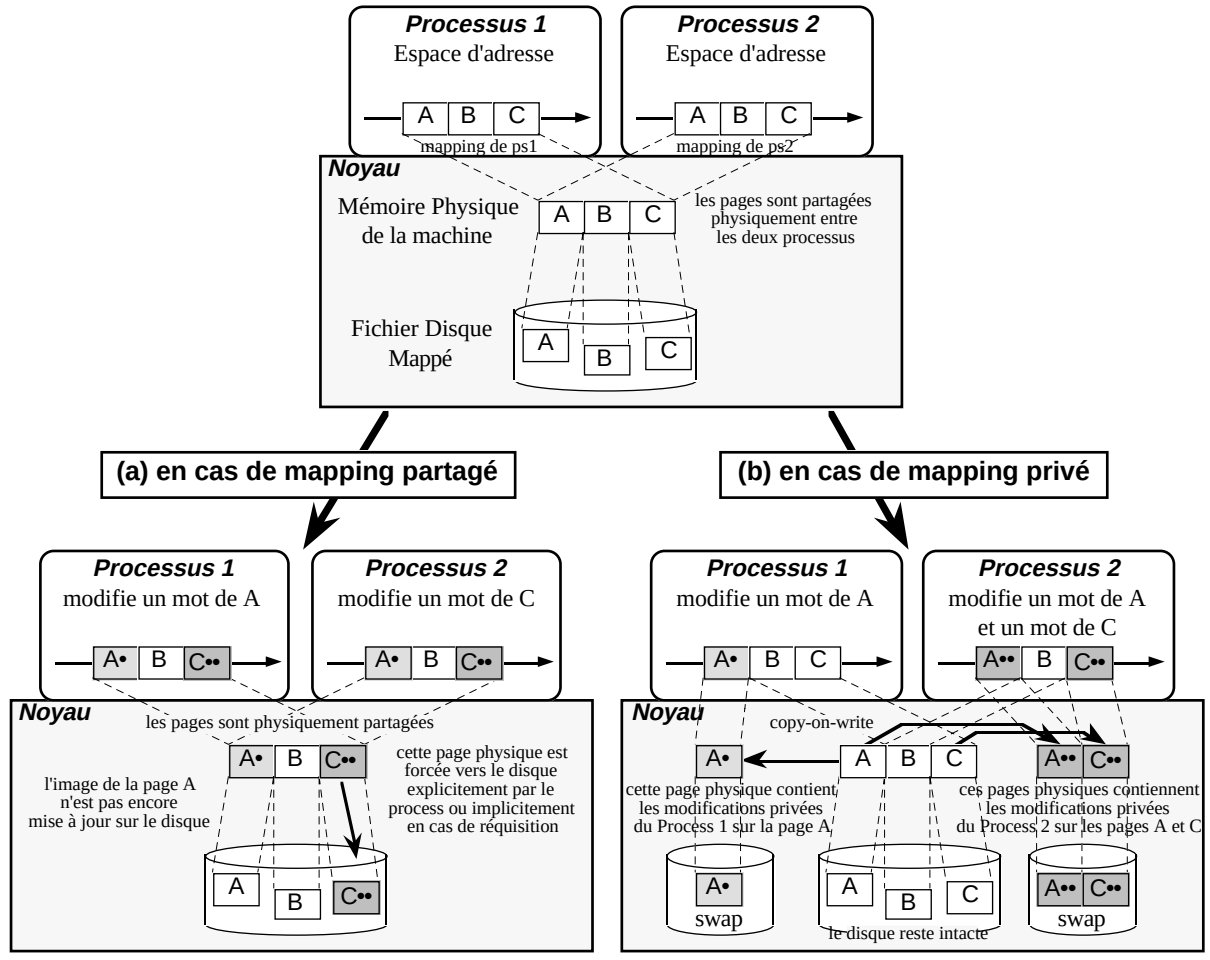
Les systèmes d'exploitation offrent désormais une alternative aux appels systèmes `read()` et `write()` pour accéder au contenu d'un fichier. Le Memory-Mapping [Abrossimov89, SunSoft93b] consiste à installer l'image du fichier (ou d'une partie du fichier) dans l'espace d'adressage de la tâche (processus). Un accès mémoire dans la zone de mapping provoque le chargement en mémoire de la page disque contenant le mot accédé. Cette page est alors gérée comme une page quelconque par le mécanisme de mémoire virtuelle qui peut par exemple "swapper" la page du fichier vers le disque pour réquisitionner la page mémoire. Ce mécanisme est similaire aux IPC Shared Memory de System V qui n'autorisent que l'installation d'objets particuliers (les Shared Memory) dans l'espace d'adressage de la tâche.

Le Memory Mapping propose deux modes de "mapping" sur un fichier. Le **mapping en mode partagé** permet à plusieurs tâches, mappant le même fichier, de partager leurs modifications; ces modifications sont inscrites à l'intérieur du fichier quand le noyau requiert la page physique ou que le processus demande au noyau de synchroniser l'image disque avec l'image mémoire. Dans l'exemple de la figure 5.2.a, comme les pages restent physiquement partagées, la modification de la page A par le processus 1 est visible du processus 2, comme la modification de la page C par le processus 2 qui est visible du processus 1. Avec le **mapping en mode privé**, les modifications ne sont visibles qu'à l'intérieur de l'espace d'adressage de la tâche. Ce mapping privé est en général implanté avec un mécanisme de "copy-on-write" [Gingell87, Nelson88, Leffler89] qui ne duplique les pages mémoire qu'au moment de leur modification. Ces pages restent partagées jusqu'à la modification. Dans l'exemple de la figure 5.2.b, la page A est modifiée à la fois par les processus 1 et 2 qui ne voient que leur propre modification. Le fichier ne contient pas ainsi les modifications apportées par les processus. Enfin, une page physique qui contient une image modifiée peut être réquisitionnée par le noyau : l'image est écrite sur un des disques de swap de la machine.

Le Memory-Mapping est renforcé par un mécanisme de protection des accès sur les pages dans l'espace d'adressage. Ce mécanisme permet de détecter des violations de ces protections quand la thread tente d'accéder à la page dans l'espace d'adressage avec mode d'opération incorrect; il renvoie alors une exception à la thread concernée [Powell91].

En plus de ces fonctionnalités, Mach fournit la notion d'objet mémoire (memory object) qui peut être servie par une tâche serveur appelée "External Pager" [Young87, Rashid87]. L'External Pager prend en charge les mouvements de données entre l'objet mémoire et le disque. Le noyau Mach et External Pager se coordonnent en communiquant via des ports.

Figure 5.2 : Mapping Privé et Partagé d'un fichier.



II.3.3. MultiThreading et Memory Mapping dans l'Espace de Travail

L'utilisation du MultiThreading n'est pas récente dans la conception de serveur. La réalisation du serveur est très simplifiée grâce à l'utilisation d'une thread comme contexte des demandes du client. Malheureusement, le MultiThreading, proposé sur certains systèmes [Kepecs85, Bach86], n'offre qu'un mécanisme de temps partagé à l'intérieur d'un processus et force le concepteur à répartir les contextes sur un pool de processus serveurs dans le cadre d'architectures parallèles. L'apparition des systèmes à micro-noyau [Armand90] ou de systèmes à noyau "monolithique" ré-entrant [Kleiman92] permettent d'utiliser le MultiThreading pour exploiter le parallélisme d'une machine.

Le parallélisme peut être obtenu :

- au niveau du traitement des différentes requêtes ou dans le traitement d'une seule requête quand la machine est un multiprocesseur,

- ou au niveau des entrées-sorties quand la machine utilise plusieurs contrôleurs (réseaux, disque, ...). Ce type d'architecture est très fréquent même sur des stations bas de gamme.

L'Espace de Travail utilise le MultiThreading pour exécuter en parallèle ou en pseudo-parallélisme les transactions applicatives ou de services. Le MultiThreading est également employé pour réaliser certaines fonctions, dans l'Espace de Travail, comme l'asynchronisme des communications ou la parallélisation des écritures dans les journaux.

Le Memory Mapping est utilisé pour construire l'espace de données de l'Espace de Travail et les espaces privés de modification des transactions. L'archive est un fichier que l'Espace de Travail installe dans son espace d'adressage. Les threads, qui exécutent les transactions, consultent et modifient les objets contenus dans les pages de l'archive en accédant à leur image dans la zone d'installation. Le mécanisme de mémoire virtuelle du noyau prend totalement en charge le chargement et le déchargement des pages de l'archive entre mémoire primaire et le disque. Le mapping privé est utilisé pour offrir un espace privé de modification pour chaque thread exécutant une transaction.

De nombreux concepteurs de Gérant d'Objets ont évoqué les faiblesses de ces mécanismes de mémoire virtuelle à usage général (general-purpose) par rapport à des méthodes spécifiques de bufferisation des pages disques [Franklin93]. Cependant, l'utilisation de la mémoire virtuelle permet au noyau d'équilibrer l'utilisation de la mémoire primaire entre le Gérant d'Objets et les autres applications de la station (GUI, deamon, ...). La mémoire virtuelle offre de nombreuses fonctionnalités pour contrôler la résidence des pages en mémoire primaire (politiques multiples d'éviction, pages virtuelles en mémoire physique, invalidation, maintien d'une page en mémoire, cartographie de la résidence). Enfin, la mémoire virtuelle et de ses mécanismes de protection sont les interfaces systèmes pour exploiter la MMU de la station : les fonctions du Gérant d'Objets comme la bufferisation des pages et le verrouillage implicite sont réalisables sans surcoût logiciel.

III. Accès aux Pages Persistantes

L'Espace de Travail est une tâche (processus) dans laquelle chaque transaction (applicative ou de service) est exécutée par une thread distincte (dite d'activité). Nous verrons en section IV le rôle exact de cette thread dans la structure multi-threadée de l'Espace de Travail. L'espace d'adressage du processus Espace de Travail est utilisé pour mapper la base afin que les threads puissent accéder aux données par de simples accès mémoire.

Cette section détaille ce mécanisme d'accès aux pages persistantes de la base; les threads d'activités exécutent le code des transactions qui ne contient pas d'appel aux fonctions suivantes du Gérant d'Objets :

- bufferisation des pages disque en mémoire,
- détection de la nature des accès pour le contrôle de concurrence,
- localisation des pages provenant de volumes locaux ou distants.

Ces fonctions sont appelées implicitement par le "matériel", ce qui évite de subir le surcoût logiciel qui est nécessaire pour tester le moment où ces fonctions doivent être appelées.

Ce mécanisme repose essentiellement sur l'utilisation des différentes facettes du Memory-Mapping offertes par le noyau du système hôte. Nous présenterons nos motivations pour concevoir l'accès aux données avec le grain de la page plutôt qu'avec le grain objet. Nous verrons ensuite les trois notions qui hiérarchisent la base. Nous détaillerons alors les mécanismes d'accès aux pages persistantes et de validation des modifications. Nous reprendrons ces mécanismes pour accéder aux pages servies par un autre Espace de Travail. Enfin nous verrons que la propriété passante et la propriété englobante des Espaces de Travail sont implantées au niveau des volumes et utilisent les mêmes mécanismes avec quelques différences.

III.1. Granularité Page

Au niveau conceptuel, la base de données se compose d'objets tandis qu'au niveau physique, elle est divisée en pages de longueur fixe qui correspondent à l'unité minimum de transfert entre la mémoire et le disque. Contrairement à la page, l'objet est l'unité logique de manipulation des données; sa taille est indépendante de la taille physique de la page. Ainsi une page peut contenir plusieurs objets ou bien un objet peut s'étendre sur plusieurs pages. L'objet et la page sont les deux granularités de données utilisées au sein d'un Gérant d'Objets. La granularité, utilisée dans les différentes fonctions des Gérants d'Objets, influence largement les performances de celui-ci. Les principales fonctions concernées sont :

- le transfert des données entre client et serveur.

  Le coût des communications entre le client et le serveur comporte généralement un coût fixe important qui rend le transfert d'une page d'objets équivalent à celui d'un objet isolé.

- le contrôle de concurrence (verrouillage).

  Le contrôle de concurrence sur les pages nécessite aussi moins de communication entre le client et le serveur et des structures de gestion plus légères que dans le cas d'un contrôle effectué au niveau des objets. Par contre, la granularité page augmente la contention entre les transactions du système. En effet en verrouillant une page, une transaction verrouille également des objets qu'elle n'accède pas; une autre transaction qui souhaite accéder à un de ces objets, doit alors se mettre attendre (dans le cas d'une méthode par verrouillage). La contention entraîne un surcoût CPU, lié aux blocages plus fréquents des transactions (ce coût correspond au changement de contexte du processus ou de la thread qui exécute la transaction).

- la gestion des réplicas (callback)

  Les clients cachent les données et les verrous en lecture associés, d'une transaction à l'autre. Une donnée peut être répliquée dans plusieurs caches; elle doit donc être invalidée dans ceux-ci, avant d'être modifiée par un client.

Les Gérants d'Objets appliquent généralement une seule granularité pour toutes ces fonctions; cependant la possibilité d'utiliser un grain adaptatif a été proposée par [Carey94b] pour adapter le grain de verrouillage et de réplication dans les caches clients au degré de contention des transactions sur les pages.

La page est aussi l'unité de mémoire virtuelle manipulée par les mécanismes matériels de MMU (Memory Management Unit) des processeurs à usage général [Hennessy90, Seznec92]. Les mécanismes de protection de ces MMU détectent la résidence d'une page en mémoire primaire ou une violation des protections[1] sur une page. Le Gérant d'Objet peut profiter de ces fonctionnalités pour réaliser le mécanisme de bufferisation des données et de détection du verrouillage. L'utilisation de la MMU rend ce mécanisme totalement transparent aux transactions. Cette approche diminue largement le coût CPU d'accès aux données en mémoire (une fois la page chargée et verrouillée).

L'Espace de Travail utilise la granularité Page pour les différentes fonctions. Ce choix a été principalement motivé par la possibilité d'utiliser les mécanismes de MMU.

III.2. Volume - Segment - Page

La granularité Page choisie pour la conception de l'Espace de Travail nous force à considérer la base de données comme un ensemble de pages persistantes réparties entre les différents Espaces de Travail du système.

Nous considérons désormais que les trois notions hiérarchiques organisent la base de données :

- le Volume,
- le Segment,
- la Page.

Le Volume représente un ensemble de pages qui se trouvent sur le même support physique. Ce support physique peut être un disque "brut" (raw) sans organisation de fichier ou un fichier (c'est à dire un ensemble de blocs disque organisé par le système de fichier du noyau). Dans le cadre des services de Données ou Coopératif, le volume est l'unité minimale qu'un Espace de Travail peut servir. Dans ce cas, l'identifiant du Volume qualifie ce service. Un volume est dit **local** à un Espace de Travail quand celui-ci accède directement à ses pages. Un volume est dit **distant** quand l'Espace de Travail est obligé de passer par le service de Données, proposé par un autre Espace de Travail, pour accéder à ces pages.

Le segment représente un ensemble de pages contiguës à l'intérieur du volume. Cette contiguïté peut être seulement logique sur disque, si le Volume est implanté par un fichier. Néanmoins, la continuité des pages est garantie quand le segment est installé en mémoire virtuelle. Nous verrons dans le paragraphe suivant que le segment correspond à l'unité d'installation en mémoire virtuelle des pages du volume (chaque page du segment est couplée à une page de la mémoire virtuelle au cours de l'installation).

---

1 Certaines MMU de processeurs proposent un grain de protection de l'espace d'adressage plus petit que la page physique : la page virtuelle, qui correspond toujours à une seule page physique, est partagée en plusieurs zones de protection. La MMU de l'ARM610, utilisé pour le Newton d'Apple [Smith94,Welland94], offrent des zones de 1K avec des protections différentes. Dans cette machine, le but de ces zones est de réduire la consommation de la mémoire primaire, mais en bases de données, une telle MMU permet de définir un verrouillage plus fin pour réduire le taux de contention, tout en conservant les avantages des MMU. Cette granularité plus faible présentera un intérêt réel si cette technique se généralise dans la prochaine génération de processeurs.

La page représente la granularité utilisée à la fois pour le transfert de données entre Espaces de Travail, pour le contrôle de concurrence et pour l'invalidation du cache client.

III.3. Accès aux Pages d'un Volume Local par les Transactions

Le Memory-Mapping est utilisé pour installer dans l'espace d'adressage de l'Espace de Travail les pages persistantes d'un volume. Les threads d'activité qui exécutent les transactions peuvent accéder au contenu de ces pages par de simples accès mémoire. Le Memory-Mapping permet de cacher à la transaction les fonctions suivantes :

- la bufferisation des pages disque en mémoire primaire,
- les modifications dans un espace privé,
- le contrôle implicite de la concurrence.

Ces fonctions sont soit entièrement prises en charge par le noyau, soit exécutées par la thread dans un mode d'exception : les appels à ces fonctions sont donc totalement absents du code des transactions.

Nous ne nous occupons dans cette section que de l'accès aux pages appartenant à un volume local à l'Espace de Travail. L'accès aux volumes distants qui utilise les mêmes techniques est traité dans la section III.4.

III.3.1. Contrôle de Concurrence par Verrouillage 2-Phase

L'implantation de l'accès aux pages est réalisée dans le cadre de l'utilisation du Verrouillage 2-Phases (Two-Phase Locking) [Bernstein81]. Cette technique de contrôle de concurrence est constituée d'une phase de pose de verrous, suivie d'une phase de dépose des verrous accordés lors de la terminaison de la transaction. La transaction demande un verrou sur un granule de donnée quand la transaction accède pour la première fois à celui-ci. Le verrou demandé peut être un verrou partageable ou un verrou exclusif suivant que l'accès est une consultation ou une modification. Le verrou peut être accordé à la transaction s'il n'y a pas de conflit ou bien être mis en attente de l'accord. Les demandes en attente sont organisées en groupe d'attente ("Pending Groups") pour éviter la famine pour les demandes de verrous exclusifs [Gruber92a]. Une transaction, mise en attente, est ajoutée au dernier groupe si sa demande est compatible avec celle du groupe; sinon, elle crée un nouveau groupe. Une fois le verrou accordé, la transaction réalise l'opération sur le granule. La transaction peut demander une augmentation du verrou partageable vers un verrou exclusif : l'augmentation peut être bloquante si d'autres transactions possèdent aussi le verrou partageable.

Dans l'Espace de Travail, le grain de verrouillage est la Page. Ce grain nous permet d'utiliser les protections mémoires des pages mappées, pour détecter implicitement les modes d'opérations (consultation ou modification) qui sont appliqués sur ces pages, par la thread qui exécute la transaction.

III.3.2. Accès aux pages au travers d'un mapping privé d'un segment

La thread accède (consulte et/ou modifie) une page persistante au travers du mapping privé du segment qui la contient. Le segment est donc l'unité de mapping dans l'Espace de Travail. Le mapping privé définit un espace privé de modification pour la thread : celle-ci peut modifier l'image d'une page persistante tout en préservant l'image originale dans le segment disque.

L'accès aux pages est réalisé de la manière suivante :

1- La thread doit préalablement mapper le segment auquel la page appartient.

> Les pages du mapping sont initialement protégées contre tout accès. L'installation du mapping ne provoque pas le chargement des pages mappées en mémoire primaire. (figure 5.3.a)

2- La thread peut alors consulter et/ou modifier un mot de la page en accédant à l'adresse correspondante dans l'espace d'adressage.

> Cette adresse se calcule à partir de l'adresse du mapping du segment, de la position de la page dans le segment et de la position du mot dans la page.

3- Le premier accès mémoire dans le mapping de la page provoque une violation des protections qui est détectée par le noyau (figure 5.3.a).

> Le noyau notifie[2] alors cette violation à la thread. Celle-ci interprète la violation comme un nouvel accès dans une page et demande donc au mécanisme de contrôle de concurrence, le verrou correspondant. La nature du verrou demandé (partagé, exclusif) dépend de la nature de la violation[3] :
>
> - Le verrou partagé (ou Read-Lock) est demandé si l'instruction fautive consulte la page (i.e. changement du registre du processeur avec un mot mémoire de la page persistante).
> - Le verrou exclusif (ou Write-Lock) est demandé si l'instruction fautive modifie la page (déchargement d'un registre vers un mot mémoire de la page persistante).
>
> Conformément au verrouillage 2-phase, la thread se met en attente du verrou s'il ne lui est pas accordé.

4- Une fois que le verrou est accordé, la page est "déprotégée" en fonction du verrou. La thread reprend ensuite le cours normal de son exécution en relançant l'instruction fautive (figure 5.3.b).

> L'image de la page est chargée en mémoire primaire seulement à ce moment. Ce chargement correspond au mécanisme de bufferisation qui est réalisé implicitement par le noyau.

---

2 Sur Solaris, la notification est réalisée par le signal SIGSEGV qui déroute la thread fautive vers un traitement d'exception. La routine d'exception demande le verrou; une fois le verrou accordé, l'exception se termine par la reprise du cours normal de la thread à partir de l'instruction fautive. BSD utilise à la fois les signaux SIGSEGV et SIGBUS pour notifier une violation.

3 Le noyau n'offre pas toujours la possibilité de déterminer la nature de la violation : Dans ce cas, le verrou demandé est celui qui est directement supérieur au verrou courant. Si celui-ci ne suffit pas, l'instruction fautive est relancée (comme dans l'alinéa 4) et elle provoque de nouveau une violation qui permet alors de demander le verrou correct.

5- Si par la suite, la thread réalise un accès dans la page qui nécessite un verrou supérieur, une violation des protections se produit (figure 5.3.b). Elle est traitée comme dans l'alinéa 3, excepté que la thread demande une augmentation du verrou : la thread reprend le cours normal de la transaction quand l'augmentation est acceptée.

L'augmentation de verrou est demandée quand l'instruction fautive est une modification du mot de la page. A la relance de l'instruction, le noyau provoque le copy-on-write de la page accédée : le mapping est réorganisé afin de préserver l'image disque originale de la page (figure 5.3.c).

Remarquons enfin que le noyau gère implicitement les chargements et les déchargements des pages en mémoire primaire. Le déchargement occasionne une écriture de l'image vers le disque de swap quand la page évincée a été modifiée et un abandon simple de l'image dans le cas contraire (figure 5.3.c).

La thread mappe (en privé) un segment chaque fois qu'elle souhaite accéder à une page appartenant à ce segment non mappé : la thread possède une table qui contient les adresses des segments qu'elle a mappées dans l'espace d'adressage de l'Espace de Travail. La table des segments constitue pour la thread un point d'entrée pour accéder aux données persistantes. Cette table est utilisée par l'opération de déréférenciation des identifiants d'objets (section V.3.2). Il existe également une table inversée qui est utilisée, quand une violation se produit, pour faire correspondre l'adresse de violation avec la page accédée.

Figure 5.3 : Utilisation du Memory-Mapping pour charger et verrouiller implicitement les pages

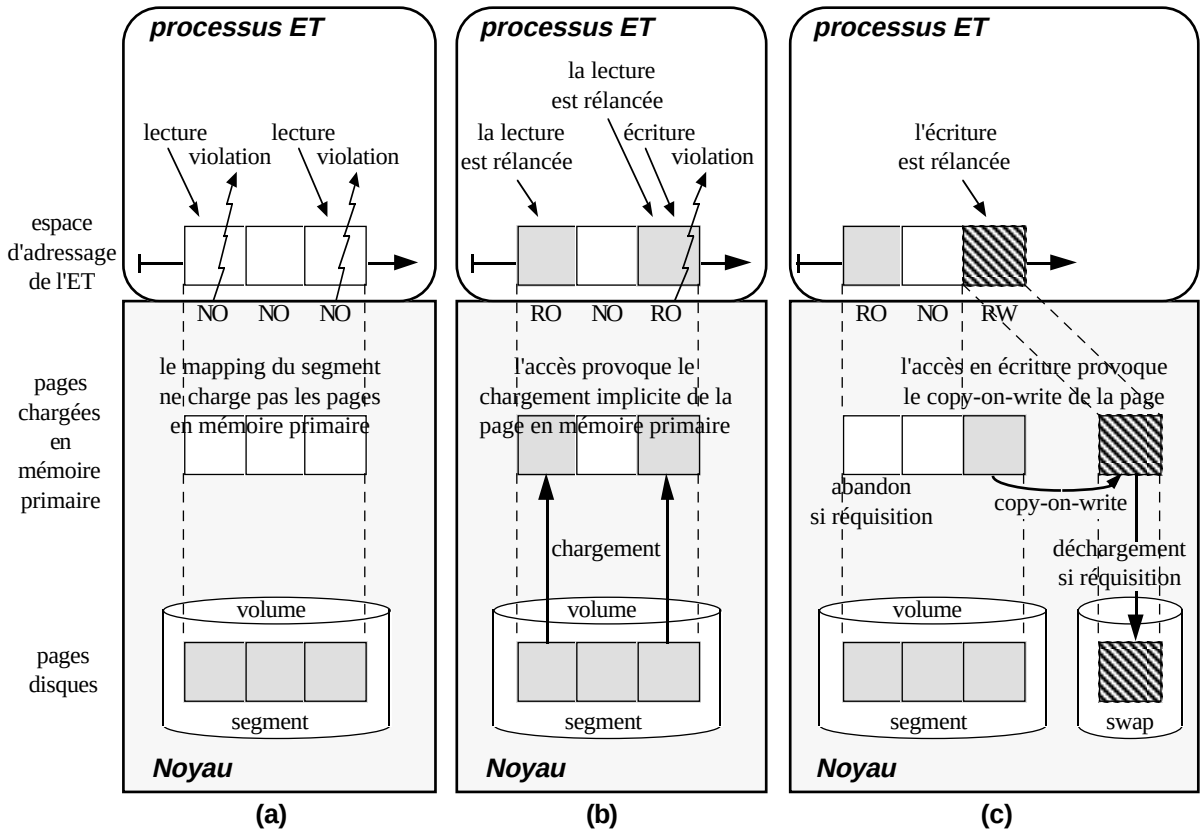
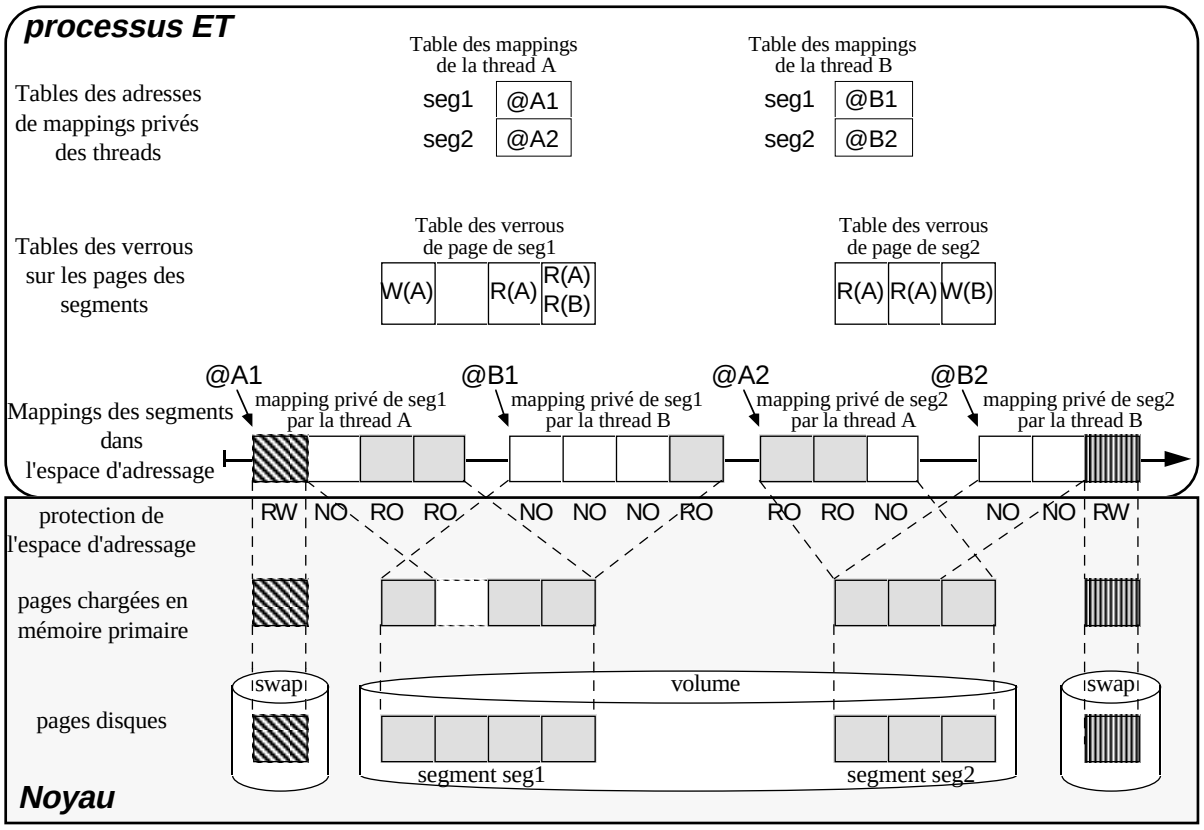


Figure 5.4 : Tables de Mappings et Tables de Verrous sur des segments.



L'Espace de Travail est le siège de plusieurs threads d'activité. Chacune d'entre elles demande des mappings privés de différents segments de la base pour accéder aux données persistantes. Chaque mapping privé est réservé à une seule thread; elle réalise ses consultations et ses modifications des pages du segment seulement au travers de ce mapping. La figure 5.4 représente un Espace de Travail qui héberge deux activités exécutées par les deux threads A et B. Les mappings A1 et A2 (resp. B1 et B2) sont réservés à la thread A (resp. la thread B) pour consulter les pages des segments seg1 et seg2 ou bien pour les modifier comme la première page de seg1 (resp. la dernière page de seg2). L'image disque des pages reste préservée de toute modification dans les mappings privés.

La concurrence est contrôlée au moyen de structures partagées par les threads[4]. Ces structures sont les tables de verrou des pages dans le cas du verrouillage 2-phase. Dans le cas d'un contrôle de la concurrence par estampillage, ces structures sont les tables d'estampilles. Dans la figure 5.4, les verrous accordés à une transaction reflètent les protections positionnées sur les pages correspondantes dans les mappings privés des threads.

L'utilisation du segment comme unité de mapping a été guidée par le coût quasi-constant[5] de l'installation d'un mapping dans l'espace d'adressage. Le coût du mapping du segment est ainsi

4 Ces structures sont gardées par des variables de synchronisation pour sérialiser leurs accès.

5 Les expérimentations des installations (`mmap()`) et des désintallations (`munmap()`) montrent que ce coût est quasi-constant par rapport à la taille du mapping sur SunOS 4.x et SunOS 5.x (Solaris 2). Valeur ....

réparti sur plusieurs pages. Cependant, l'utilisation du segment pose le problème de la consommation de l'espace d'adressage disponible : en mappant un segment, la thread mappe aussi des pages qui ne seront pas accédées au cours de la transaction. Ces emplacements non utilisés ne peuvent être récupérés en cas de saturation de l'espace d'adressage. Ce dernier point est crucial avec les processeurs 32 bits dont l'espace d'adressage est donc limité à 4Go. La quantité de pages accédées dans la base ne pourra dépasser cette limite. La taille du segment est fixée lors de sa création et peut être différente d'un segment à l'autre. La taille choisie doit être un compromis entre le coût d'installation du mapping et la consommation de l'espace d'adressage.

III.3.3. Validation des Modifications d'une Thread d'Activité.

Une thread d'activité qui termine l'exécution d'une transaction procède à la validation des modifications réalisées dans le cadre de cette transaction. La validation doit :

- rendre visible les modifications auprès des autres transactions,
- propager celles-ci vers le volume (le disque).

Le copy-on-write optimise l'utilisation de la mémoire primaire en partageant physiquement les pages non modifiées entre tous les mappings d'un fichier. Si un de ces mappings est partagé et qu'une modification est réalisée à travers lui, alors la modification reste partagée entre tous les mappings. Nous utilisons cette propriété de partage pour réaliser la validation des pages modifiées. Le mécanisme de validation requiert donc un mapping en mode partagé de chaque segment. La validation consiste à copier le contenu des pages modifiées depuis leur emplacement dans le mapping privé vers leur emplacement dans le mapping partagé. Cette copie mémoire vers le mapping partagé permet à la fois :

- de rendre visibles les modifications dans les autres mappings privés appartenant aux autres threads.

  Une des ces thread peut alors accéder à la nouvelle valeur depuis son mapping privé (après avoir demandé le verrou et levé la protection).

- de propager les modifications vers le segment disque,

  La page, modifiée dans le mapping partagé, est susceptible d'être déchargée vers le fichier en cas de réquisition de la mémoire primaire ou en cas de purge volontaire.

La validation des modifications d'une transaction réalise les trois opérations suivantes :

1- la journalisation de la page modifiée,

   Les modifications sont journalisées sur un support stable avant d'être inscrites sur l'archive (figure 5.5.b); cette condition correspond à la règle du Commit. En cas de panne, elle permet de défaire les transactions qui n'ont pas été jusqu'au bout de la validation (undo) et de reconstruire l'archive avec les modifications apportées par les transactions validées (redo) [Haerder83].

   Les enregistrements de journalisation contiennent, soit l'image complète de la nouvelle page, soit la différence entre l'image originale et la nouvelle image. La différence des deux images (ou **diffing**) permet de réduire le trafic vers le journal en groupant plusieurs enregistrements dans une page du journal. Nous proposons en section V.4. de tenir compte du contenu des pages modifiées (des objets) pour

réaliser cette opération de différence. L'enregistrement de l'image complète de la page permet de réduire la durée de la reprise. En section IV.3, nous proposons de paralléliser les journalisations dans l'Espace de Travail en pilotant plusieurs journaux parallèles (sur des médias distincts) avec un pool de threads de journalisation.

2- la copie de l'image modifiée vers le disque,

Comme nous venons de le voir, la copie de l'image d'une page modifiée, de l'emplacement privé vers l'emplacement partagé, rend la modification physiquement visible des autres threads. La copie d'une page est réalisée une fois la journalisation effectuée pour la page.

3- la dépose du verrou correspondant.

La validation correspond aussi à la phase de déverrouillage, qui réalise la dépose des verrous détenus par la transaction (figure 5.5.b). Les verrous partagés (Read Lock) sont tous relâchés en début de validation afin de limiter l'attente des autres transactions sur ceux-ci. Par contre, les verrous exclusifs (Write Lock) ne sont relâchés qu'après la journalisation et la copie de l'ensemble des pages modifiées. Les pages sont alors physiquement et logiquement visibles par les autres transactions. Elles peuvent demander les verrous pour consulter ou modifier la nouvelle valeur des pages (figure 5.5.c).

Inconvénients du mapping en mode partagé dans les validations

Le mapping en mode partagé présente trois inconvénients dans la réalisation des validations ;

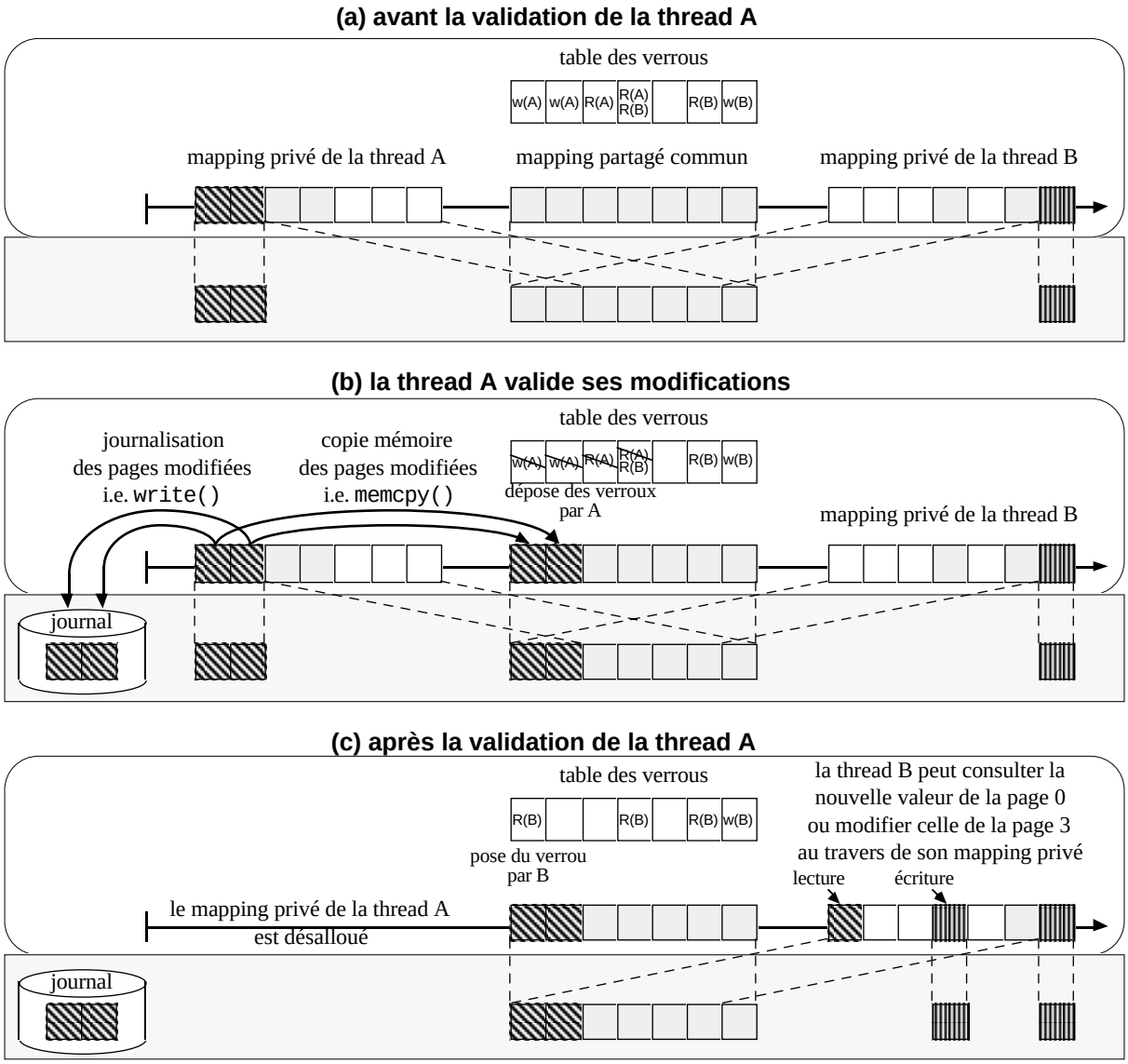
- la probabilité du chargement inutile de la page originale en mémoire,
- la réalisation d'un transfert mémoire par copie,
- la consommation de l'espace d'adressage de l'Espace de Travail.

Le premier inconvénient se produit quand l'image de la page originale n'est plus en mémoire primaire à cause des réquisitions. Le transfert par copie mémoire de l'image modifiée, depuis le mapping privé vers le mapping partagé, force l'image originale à revenir en mémoire primaire. Ce transfert disque vers mémoire est en fait inutile car l'image originale est de toute manière écrasée par la copie.

Cet inconvénient pourrait être résolu en écrivant directement la page modifiée vers le disque par l'appel système `write()` sans passer par le mapping partagé. Malheureusement sur Solaris 2, le Memory-Mapping et les accès explicites aux fichiers (appels `read()` et `write()`) sont des mécanismes qui produisent des incohérences entre l'image mémoire et l'image disque des pages s'ils sont utilisés concurremment sur une même portion de fichier [Kleiman92 §3.2].

Nous verrons dans la section IV.4 que le transfert disque vers mémoire de l'image originale n'est pas inutile si la journalisation l'utilise pour enregistrer les différences entre les deux pages.

Figure 5.5 : Validation des modifications d'une thread par Memory-Mapping

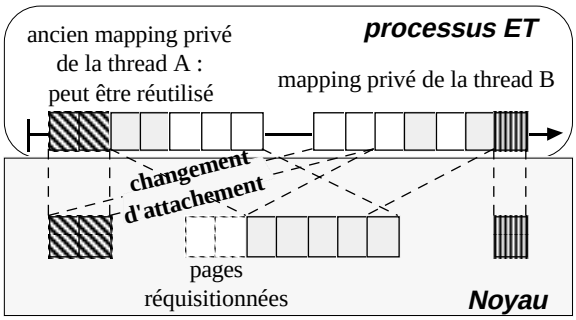


Le deuxième inconvénient est que le transfert mémoire de l'image modifiée vers l'image originale est réalisé par une copie mémoire.

Cette copie (qui est réalisée en mode utilisateur) pourrait être optimisée par le noyau en effectuant un simple attachement des mappings privés des autres threads sur la page modifiée. Cette approche est utilisée notamment par les concepteurs de systèmes à micro-noyau, pour transférer un message d'une tâche à l'autre [Condict93]. La page qui contenait l'image originale pourrait être alors réquisitionnée si elle était encore en mémoire.

Cette approche éviterait également la désallocation du mapping privé quand une transaction se termine. Ce mapping pourrait être réutilisé pour une autre transaction de l'Espace de Travail accédant au même segment.

Figure 5.6 : Partage par ré-attachement des pages modifiées



La figure 5.6 reprend les mappings de la figure 5.5 : les 2 premières pages virtuelles du mapping privé de B, sont attachées aux 2 pages physiques modifiées par la thread A. Les images originales sont abandonnées et leurs pages physiques réquisitionnées. Malheureusement, cet attachement ne peut être réalisé que par le noyau.

Le troisième inconvénient concerne la consommation de l'espace d'adressage par les mappings partagés.

> L'utilisation augmente la consommation de l'espace d'adressage de l'Espace de Travail. Cependant, les mappings partagés peuvent être désalloués en dehors des périodes de validations, quand il faut réquisitionner de la place dans l'espace d'adressage pour d'autres mappings privés ou partagés. Nous verrons que le mapping partagé d'un segment est également utilisé, quand une page provenant d'un serveur, est chargée par l'Espace de Travail client (cf. section III.4.3), ou quand une page est accédée pour la première fois depuis le démarrage de l'Espace de Travail pour réinitialiser les pointeurs vers les méthodes virtuelles des objets de cette page (cf. section VI.2.2). Ce type de politique d'allocation et de désallocation est très similaire à la solution prise par le système MASIX, pour accéder au contenu d'un fichier au travers d'une fenêtre de mapping [Card94].

III.4. Accès aux Pages d'un Volume Distant par les Transactions

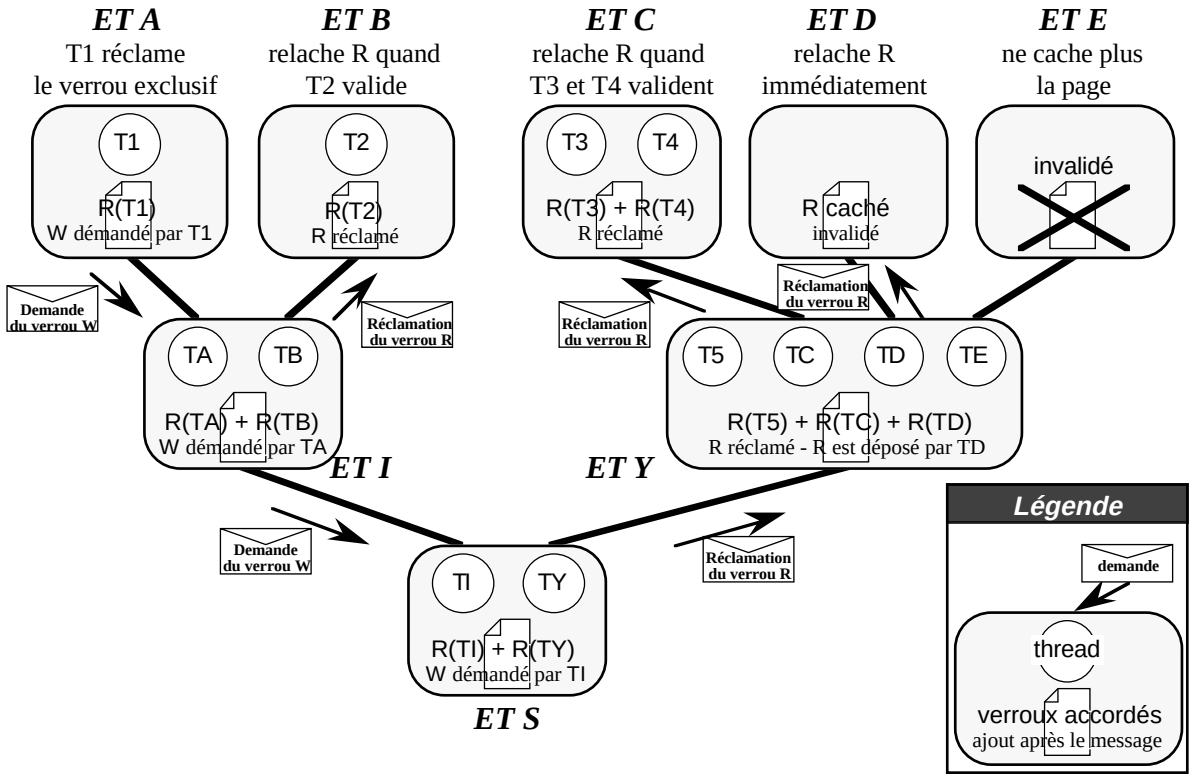
L'accès aux pages distantes est réalisé dans le contexte du service de Données ou d'un service Coopératif portant sur le Volume contenant les pages demandées. Dans ce contexte, le Verrouillage par Rappel (Callback Locking) a été adapté aux hiérarchies d'Espaces de Travail, pour contrôler la concurrence des accès. Dans les Espaces de Travail abonnés, les threads utilisent aussi le Memory-Mapping pour accéder aux données.

III.4.1. Contrôle des Accès Concurrents avec le Verrouillage par Rappel.

La construction hiérarchisée des architectures d'Espaces de Travail nous a amené à utiliser le Verrouillage par rappel afin de cacher des données partagées entre les différents Espaces de Travail tout en maintenant la constitance transactionnelle.

Dans une architecture bases de données Client-Serveur, le client cache les pages (les objets) demandées par les transactions dans sa mémoire locale. Lors de la validation d'une transaction, les pages cachées dans le client sont soit invalidées du cache (intra-transaction caching), soit conservées en vue d'être réutiliser par une transaction suivante (inter-transaction caching) du client. Dans une implantation classique du verrouillage 2-phase, les pages du cache sont

Figure 5.7 : Demande de verrou exclusif dans une hiérarchie d'Espaces de Travail.



invalidées après chaque validation; et donc une page, accédée par deux transactions successives, est demandée deux fois (avec le verrou associé) au serveur[6].

Le Verrouillage par Rappel [Wilkinson90, Wang91, Franklin93] conserve la page et le verrou partagé sur le client, pour que d'éventuelles transactions successives puissent y accéder sans demande auprès du serveur. En revanche, lorsqu'un verrou exclusif est déposé, le client ne cache pas ce verrou mais conserve néanmoins celui-ci en mode partagé[7]. Ce protocole implique que plusieurs clients peuvent cacher le verrou partagé; la demande du verrou exclusif par un client réclame le relâchement du verrou partagé par les clients qui le cachent. Le relâchement du verrou partagé est réclamé à ces clients par le serveur : le client relâche le verrou si toutes les transactions, détentrices du verrou, ont terminé leur validation. La réclamation est donc traitée prioritairement par rapport aux autres demandes en attente (Pending Groups) : les transactions, qui restent en attente, renouvellent la demande de verrou au serveur. Sur un réseau local, le serveur peut diffuser l'ordre de relâchement par un multicast ou broadcast [Liang90]. Le protocole de Verrouillage par Rappel est semblable au protocole *write-invalidate* de gestion de cache dans les multiprocesseurs [Stenström90].

---

6 Sur Exodus, les pages restent cachées sur le client; cependant les verrous sont demandés au serveur à chaque nouvelle transaction. Dans le contexte d'un réseau, le transfert d'un verrou seul a sensiblement le même coût que le transfert d'un verrou accompagné de l'image de la page.

7 Le Verrouillage par Rappel n'est en réalité pas lié à la nature des verrous cachés par les clients : un client pourrait fort bien cacher des verrous exclusifs. Cependant, cacher le verrou exclusif demande un grand nombre de messages de réclamation dans un environnement "normalement" concurrent [Wang91]. Dans le cadre d'un environnement peu concurrent, le serveur pourrait autoriser le client à cacher le verrou exclusif, tant qu'il est seul à répéter les demandes pour ce verrou.

Dans le contexte hiérarchique des Espaces de Travail, un Espace de Travail intermédiaire réagit à la fois comme un client et un serveur dans le cadre des relâchements :

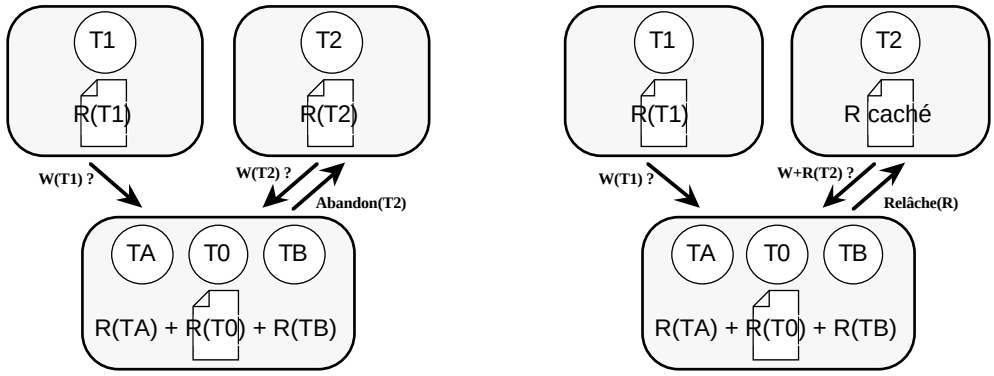
- quand une demande de verrou exclusif provient d'un des clients, l'Espace de Travail intermédiaire agit comme un serveur en diffusant la réclamation vers les autres clients, et comme un client en demandant le verrou exclusif au serveur. Dans la figure 5.7, quand la transaction T1 de A demande le verrou exclusif, l'intermédiaire I se trouve dans ce cas. Il relaye la demande vers son serveur S et rappelle le verrou à l'Espace de Travail B.
- quand une demande de rappel est réclamé par le serveur, l'Espace de Travail intermédiaire agit comme un serveur en diffusant la réclamation vers tous ses clients; puis comme un client en relâchant le verrou une fois que tous ses clients l'ont relâché. L'Espace de Travail intermédiaire Y de la figure 5.7 se trouve dans ce cas. Il diffuse le relâchement à ses clients C et D. Il relâche ensuite le verrou vers le serveur, quand la transaction T5 a validée et que le verrou a été relâché par C et D.

Il existe un conflit quand le serveur reçoit simultanément[8] deux demandes du verrou exclusif. Ce type de conflit regroupe en fait deux cas d'accès aux données par les deux transactions conflictuelles :

- Deux transactions consultent déjà la donnée (i.e elles possèdent donc le verrou partagé) et souhaitent la modifier (figure 5.8.a). C'est un cas d'interbloquage et une des deux transactions doit être abandonnées.
- Une des transactions souhaite modifier la donnée mais ne possède pas le verrou partagé. (figure 5.8.b) Il n'y a pas d'un conflit réel si le verrou exclusif est accordé à la transaction qui possède déjà le verrou partagé et que l'autre est mise en attente.

Pour connaître exactement la nature du conflit en présence, le serveur distingue deux types de demandes de verrous exclusifs : celui des transactions qui possèdent déjà le verrou partagé et celui qui ne le possède pas.

Figure 5.8 : Conflits de Demande du Verrou exclusif



(a) Conflit Write / Write **(b) Conflit Write / Read-Write**

8 Ce cas se produit quand un des clients émet sa demande de verrou exclusif vers le serveur, avant de recevoir de celui-ci la demande de réclamation du verrou partagé.

III.4.2. Accès à une Page Distante au travers du Segment Image

Le mécanisme d'accès aux pages offre un accès uniforme aux pages provenant du volume local ou aux pages d'un volume distant. Pour accéder à une page locale, une thread mappe en mode privé le segment de ce volume qui contient la page. Dans le cas d'une page distante, la thread ne mappe pas directement le segment distant[9] qui peut se trouver sur une autre machine; elle mappe un segment image qui contient des images importées des pages du segment distant. L'importation des pages est réalisée par le service de Données (ou Coopératif) du volume dans lequel se trouve le segment.

Le segment image est donc un disque (ou un fichier) local à l'Espace de Travail. Sa taille est celle du segment distant et son contenu est vide à l'initialisation de l'abonnement auprès de l'Espace de Travail serveur. Une thread qui souhaite accéder à des pages du segment distant, mappe en privé le segment image. Comme dans le cas du segment local, ce mapping est initialement protégé contre tous les accès. Le premier accès dans une page de ce mapping provoque une violation des protections mémoire. Le noyau notifie la thread qui interprète la violation comme un nouvel accès à une page du segment distant. La thread se trouve face à différents cas exposés dans le Verrouillage par Rappel:

- l'image de la page distante n'est pas cachée dans le segment image

  La page est demandée avec le verrou correspondant au serveur à moins qu'une thread ait déjà demandé ceux-ci. Elle se met en attente du retour de la page et du verrou avec la thread déjà en attente.

- l'image de la page distante est présente dans le segment image

  la thread se retrouve dans le cas de la gestion locale du verrou excepté pour le verrou exclusif qui doit être demandé au serveur.

Ce segment image est local à l'Espace de Travail; son contenu est constitué par l'importation de pages, réclamées par les threads, au moyen du service de Données portant sur le volume distant. La figure 5.9 illustre l'importation d'une page au travers de ce service :

1- La thread A tente d'accéder à la page 4 dans son mapping privé et viole les protections. Comme la page 4 n'est pas présente dans le segment image, la thread demande l'image de cette page accompagnée du verrou au serveur.

2- Coté serveur, la thread qui exécute une activité de service accède à la page 4 comme une thread quelconque : elle demande le verrou local à l'Espace de Travail serveur. Une fois que le verrou est accordé, elle consulte la page pour l'envoyer au client.

3- Une thread particulière reçoit sur le client la page remontée par le serveur. Cette thread dite d'Abonnement installe la page dans le segment image; elle copie cette page depuis le buffer de communication vers son emplacement dans le mapping partagé du segment image. Enfin, la thread réveille les threads en attente du verrou et de la page, après leur avoir accordé celui-ci dans la table locale des verrous. Nous verrons dans la section

---

9 Mapper un fichier distant au travers de NFS est possible mais dans le cadre d'un gérant d'objet, cette solution a l'inconvénient d'avoir un coût double de communication : le verrou et l'image de la page voyagent dans des messages séparés.

IV.2.1 que cette thread est dédiée à l'abonnement vers le serveur, pour rendre les communications asynchrones entre le client et le serveur.

III.4.3 Validation des Modifications d'un Segment Distant.

La validation des modifications sur un segment distant reste très semblable à celles sur un segment local.

1- Journalisation des pages modifiées vers l'Espace de Travail serveur,

La thread envoie au serveur un enregistrement de journalisation contenant la différence entre la page originale et la page modifiée. Plusieurs enregistrements sont bufferisés dans une page avant d'être transmis au serveur.

De son côté, la thread serveur réalise une validation locale avec les enregistrements reçus : elle journalise directement les enregistrements reçus et applique ensuite les différences, contenues dans les enregistrements de journalisation, sur les pages du segment au travers du mapping partagé de l'Espace de Travail serveur.

2- Visibilité des pages modifiées dans l'Espace de Travail,

Comme dans le cas des modifications sur une page locale, le transfert, de l'emplacement privé vers l'emplacement partagé, est utilisé pour rendre une page visible des autres threads de l'Espace de Travail.

Cette étape n'est pas réalisée dans le cas où le thread serveur a demandé au client l'invalidation du verrou partagé.

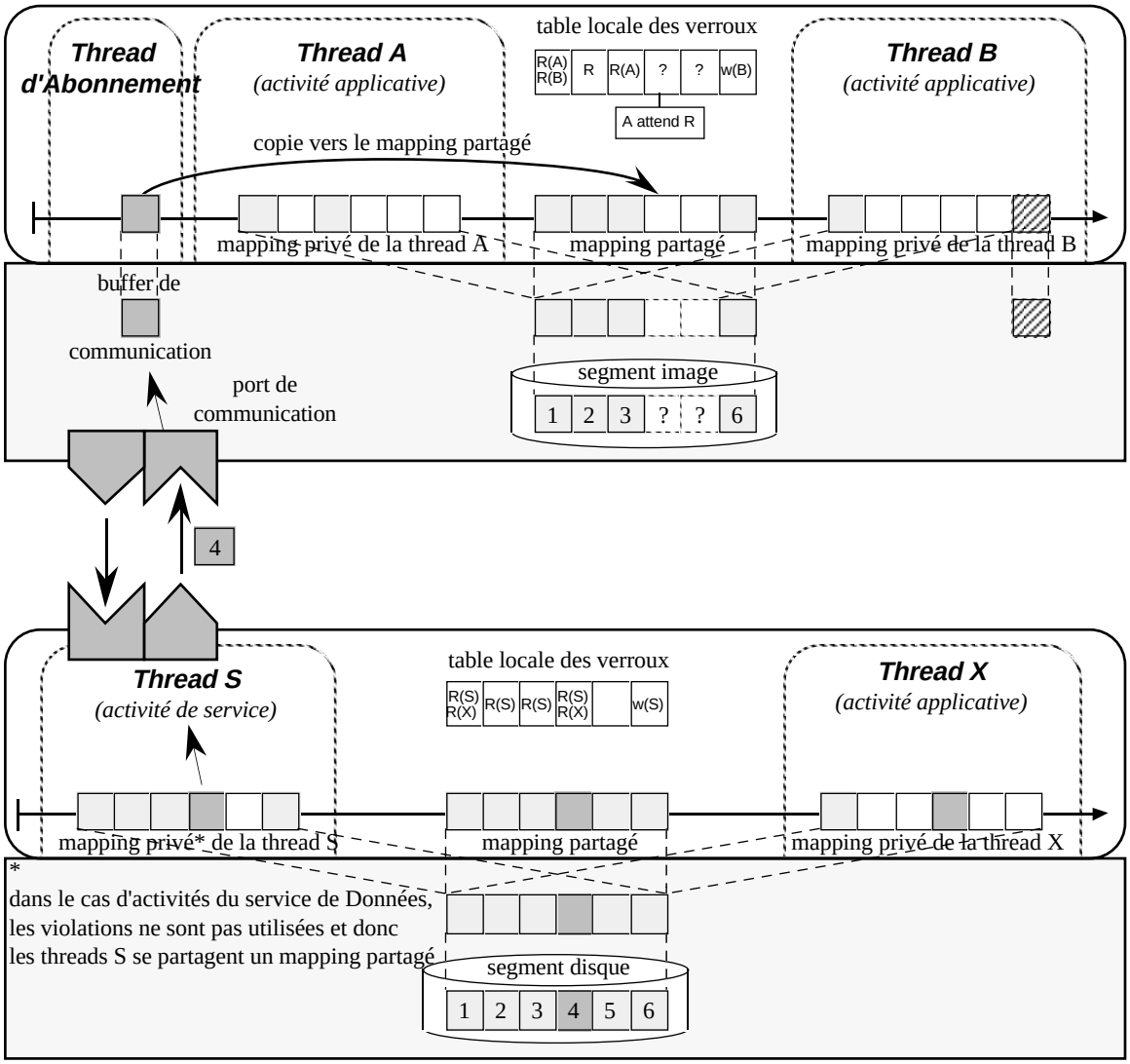
3- Déverrouillage des pages accédées.

La validation se termine par la dépose des verrous accordés à la transaction. Conformément au Verrouillage par Rappel , si le verrou accordé est exclusif, la thread relâche ce verrou auprès du serveur et le transforme en verrou partagé : l'image de la page et le verrou partagé restent ainsi cachés sur le client. Cependant, si le serveur réclame au client le verrou (donc l'invalidation de la page), le client ne cache pas le verrou partagé quand la dernière transaction détentrice se termine.

Dans la validation distante, il n'est pas nécessaire que le client respecte le protocole WAL (du moment qu'il est respecté par l'activité serveur) : le client n'attend pas la confirmation de la journalisation effective sur le serveur pour copier les modifications sur le segment image. Si la validation n'est pas achevée (abandon), les modifications sont défaites du segment image par une simple invalidation des pages modifiées dans celui-ci. Cette technique est aussi utilisée par l'Espace de Travail englobant (section III.5)

La validation montre bien que côté serveur, la thread serveur (du service de Données ou Coopératif) n'effectue que des consultations dans son mapping privé. Nous utilisons cette propriété pour optimiser l'utilisation de l'espace d'adressage en affectant le mapping commun aux threads des services de Données et Coopératifs (cf. IV.2.3).

Figure 5.9 : Accès à une page d'un volume distant



III.5. Gestion des Volumes dans un Espace de Travail Englobant

Les mécanismes décrits dans les sections précédentes réalisent une gestion passante des volumes locaux ou distants. Cette section propose de réutiliser ces mécanismes (et de les adapter) pour effectuer la gestion englobante de ces volumes.

III.5.1. Rappel de la propriété Englobante

Dans la section II.2.4 du chapitre 3, nous avons défini la propriété K d'un Espace de Travail. Cette propriété détermine le mode de validation de modifications apportées par les transactions à l'intérieur d'un Espace de Travail.

- Le mode passant propage les modifications d'une transaction vers l'archive et vers les serveurs.

- le mode englobant considère les transactions de l'Espace de Travail comme des transactions imbriquées à l'intérieur d'une transaction englobante. Les modifications apportées par les transactions "imbriquées" sont communes à celles-ci; toutefois elles ne sont validées que lorsque la transaction englobante valide les modifications apportées par ses transactions "imbriquées".

Dans notre implantation, la propriété K n'est pas une propriété globale de l'Espace de Travail; chaque volume, utilisé par l'Espace de Travail, peut être géré de manière indépendante suivant le mode englobant ou suivant le mode passant. Un Espace de Travail englobant (resp. passant) est en réalité un Espace de Travail dans lequel tous les volumes sont gérés sur le mode englobant (resp. passant).

III.5.2. Gestion Englobante d'un Volume Distant

La gestion englobante du volume distant est identique de la gestion passante pour la demande (importation) de verrous et/ou de pages. Les différences entre les deux gestions portent sur la validation des transactions "imbriquées" par l'Espace de Travail, sur la gestion des rappels, et bien entendu sur les validations et abandons globaux qui n'existent pas dans la gestion passante.

Lorsqu'une transaction de l'Espace de Travail englobant valide des modifications sur un segment distant, elle recopie les modifications apportées vers le mapping partagé du segment image associé et dépose le verrou exclusif localement dans l'Espace de Travail. Toutefois, elle n'envoie pas d'enregistrements de journalisation vers le serveur et conserve le verrou exclusif caché sur le client.

La gestion du contrôle de concurrence est différente du point de vue du rappel. En effet, l'Espace de Travail conserve un verrou (partagé ou exclusif) caché jusqu'à la validation globale (ou abandon global); un rappel sur ce verrou n'est satisfait qu'au moment de la validation globale.

L'ordre de validation globale consiste à redescendre l'image complète des pages modifiées depuis la dernière validation globale. Notons qu'il n'est plus possible de construire les enregistrements de journalisation à partir de l'image originale de la page (i.e. technique de "diffing" section V.4) car celle-ci n'existe plus sur l'Espace de Travail client; cette opération est réalisée sur le serveur lors de la journalisation des pages redescendues (cf. section IV.3). La validation globale relâche également chaque verrou exclusif et le transforme en verrou partagé à moins que un rappel ait été demandé sur ce verrou. L'ordre d'abandon global consiste simplement à invalider les pages modifiées depuis la dernière validation globale en relâchant les verrous correspondants.

III.5.3. Gestion Englobante d'un Volume Local

La gestion en mode englobant d'un volume local est très semblable à la gestion en mode passant; la gestion englobante ajoute les ordres de validation globale et d'abandon global à la gestion passante. Nous proposons deux manières d'implanter la gestion en mode englobant d'un volume local.

La première proposition consiste à laisser les transactions (imbriquées) de l'Espace de Travail à valider leurs modifications directement sur le volume local : une transaction journalise

ses modifications, puis écrit celles-ci sur le volume local au moment de la validation. L'ordre de validation globale est immédiat puisque les modifications sont déjà sur l'archive; il ajoute dans les journaux l'indication de la validation globale. En revanche, l'ordre d'abandon global consiste à défaire les modifications validées par toutes les transactions depuis la dernière validation globale. Cette opération nécessite de gérer un journal des images avant.

La seconde solution utilise le mécanisme du segment image de la gestion des volumes distants. Les threads mappent, en mode privé, un segment image pour accéder aux pages d'un segment du volume local. Ce segment image est complété à travers un mapping partagé commun aux threads, au fur et a mesure des demandes, avec les pages lues depuis le segment local. L'ordre de validation globale consiste à écrire les pages modifiées du segment image vers leurs emplacements dans le segment local. La lecture et l'écriture du segment local peuvent passer par les appels systèmes `read`() et `write`() sans bufferisation au niveau du noyau; on évite de "consommer" l'espace d'adressage en n'éviter d'utiliser un mapping. L'ordre d'abandon global consiste à invalider les pages modifiées depuis la dernière validation de l'Espace de Travail. Cette proposition ne présente qu'un léger surcoût en écriture vers le disque en cas de validation globale[10] et un coût nul en cas d'abandon. Cependant, elle nécessite une importante réserve d'espace disque sur le client pour stocker les segments images.

IV. Structure MultiThreadée des Espaces de Travail

La structure de l'Espace de Travail doit répondre aux trois exigences suivantes :

- Exécuter simultanément plusieurs activités.

  Ces activités peuvent effectuer les composantes concurrentes d'une application (les outils d'un intégré) ou piloter chacune l'abonnement d'un client à un service.

- Réaliser des communications asynchrones et déséquencés entre les Espaces de Travail.

  Cette exigence découle de la précédente car les activités de l'Espace de Travail peuvent demander simultanément des requêtes aux serveurs : une activité de l'Espace de Travail doit pouvoir émettre une requête vers le serveur sans attendre que le serveur ait répondu à la requête d'une autre activité du serveur. L'ordre des réponses peut être déséquencié par rapport à l'ordre d'émission des requêtes. Par exemple, pour le service de Données, le serveur peut retarder l'exportation d'une page bloquée par le contrôle de concurrence et exporter une page libre demandée après la page bloquée.

- Introduire le parallélisme dans l'opération de journalisation.

  La régle du Commit de reprise sur panne impose que les modifications soient écrites sur un support stable (le journal) avant d'être répercuté sur l'archive. Afin d'améliorer le débit d'écriture vers le journal, celui-ci est constitué de plusieurs

---

10 Si N transactions modifient une page entre deux validations de l'Espace de Travail, le coût est de N+1 au lieu de N (i.e. N écritures vers le segment image + 1 écriture vers le segment local).

médias physiquement distincts. Cette démarche permet de paralléliser les écritures entre les différents médias. Ainsi, plusieurs pages modifiées par une ou plusieurs transactions peuvent être écrites simultanément dans les journaux.

La structure de l'Espace de Travail est donc multi-threadée et se constitue des types de threads suivants :

- les threads d'activité,

  Elles exécutent le code des composantes d'une ou plusieurs applications (thread applicatives) ou le code associé aux services (threads de service). Elles accèdent aux données suivant la méthode décrite dans la section III.

- les threads de communication,

  Elles sont invisibles du programmeur d'application et rendent asynchrones les communications entre les Espaces de Travail. Coté client, il existe une thread d'abonnement qui re-dirige les réponses du serveur vers les threads d'activité concernées. Coté serveur, la thread d'écoute attend les demandes du client à la place de la thread de service gérant son abonnement.

- les threads de journalisation,

  Elles composent l'organe de journalisation qui les utilisent en pool pour remplir les buffers avec le diffing des pages à valider et pour écrire ces buffers vers les journaux.

La première section décrit la structure d'un Espace de Travail qui n'engage aucune communication avec l'extérieur, ni en tant que client, ni en tant que serveur. Nous verrons ensuite que dans le cadre d'un service, le client et le serveur s'équipent de threads supplémentaires pour réaliser les communications asynchrones. Enfin, nous proposons une structure multi-threadée pour l'organe de journalisation qui pilote plusieurs journaux sur des médias physiques distincts.

IV.1. Structure d'un Espace de Travail Isolé

L'Espace de Travail isolé n'engage aucune communication liée à un service avec un autre Espace de Travail (client ou serveur). Il exécute une application sur des données situées dans des volumes locaux. Cette application se compose d'une ou plusieurs activités effectuées par des threads applicatives. Chaque thread exécute la suite séquentielle de transactions qui composent le code de l'activité.

La structure multi-threadée suivante est utilisé pour contrôler la concurrence des accès entre les transactions des différentes activités. Chaque thread est munie d'une file privée de messages vers laquelle les autres threads de l'Espace de Travail peuvent émettre des messages. Les files de messages constituent un moyen de communication interne à l'Espace de Travail. Elles sont utilisées ici pour relancer les threads bloquées par le contrôle de concurrence :

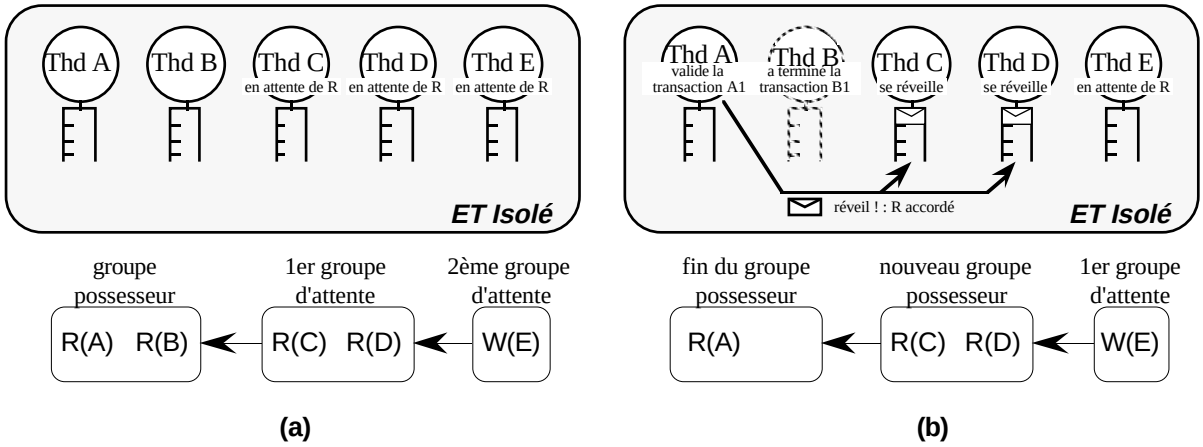
- Quand une thread souhaite accéder à une page pour laquelle elle n'a pas le verrou,

  elle consulte le verrou dans la table des verrous et si elle constate que le verrou est déjà accordé à une autre thread ou à un groupe de threads, la thread se met en attente d'un message dans sa file. Dans la figure 5.10.a, la thread E, qui demande le verrou exclusif, se met en attente dans un nouveau groupe.

- Quand une thread dépose un verrou au moment de la validation,

  elle dépose le verrou dans la table des verrous. Si celle-ci est la dernière du groupe qui possède le verrou, elle réveille les threads du groupe d'attente suivant en envoyant à chacune d'elles un message dans leur file privée de message. Dans la figure 5.10.b, la thread A valide et dépose le verrou R. Comme c'est la dernière du groupe, elle réveille les threads C et D du groupe suivant en leur envoyant un message. La thread E sera réveillée quand C et D termineront leur transaction.

Figure 5.10 : Réveil de threads en attente d'un verrou.



IV.2. Hiérarchie de Clients et Serveurs

Les hiérarchies d'Espaces de Travail se composent à la fois de clients et de serveurs mais aussi d'intermédiaires entre eux (cf. chapitre 3 section IV.2.2). Dans la plupart des services, les communications sont asynchrones et sans séquencement des réponses entre ces Espaces de Travail. Nous étendons la structure multi-threadée de l'Espace de Travail isolé pour que :

- le client puisse recevoir les réponses des demandes effectuées par ses threads applicatives.
- la thread de service puisse écouter à la fois des demandes du client et des messages internes (réponse d'un serveur ou réveil par une autre thread locale) sur l'Espace de Travail serveur ou l'Espace de Travail intermédiaire.

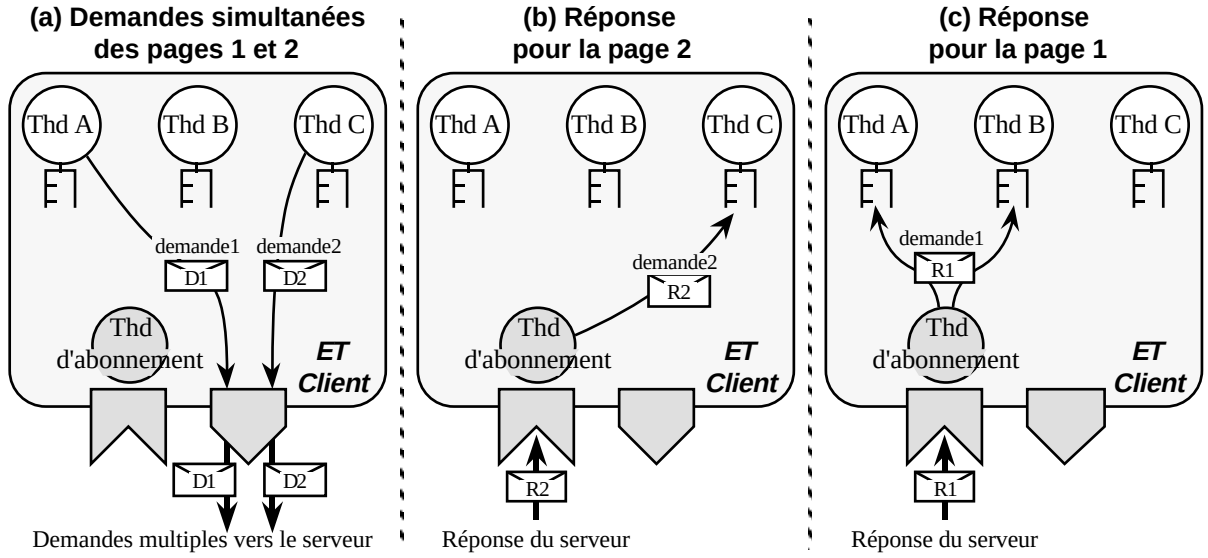
IV.2.1. Abonnement à un serveur

L'abonnement d'un client à un service met en place un canal de communication bidirectionnel entre le client et son serveur. Ce canal est réalisé par les mécanismes systèmes classiques comme les Sockets de BSD [Stevens90], les TLI de Solaris [SunSoft91] ou les Ports de Mach [Draves89]. Dans le cadre d'un Espace de Travail client, ce canal de communication est partagé par toutes les threads applicatives : les threads peuvent envoyer plusieurs demandes (i.e. requêtes) sur ce canal unique en direction du serveur; le serveur répond à chacune de ces threads par ce canal dans un ordre qui n'est pas forcément celui de l'émission.

L'Espace de Travail client est munie d'une thread d'abonnement dont le rôle est de router les réponses, reçues du canal de communication, vers les threads concernées. Une thread d'activité, qui souhaite envoyer une demande vers le serveur, émet un message contenant la demande sur le canal de communication. Elle se met ensuite en attente de la réponse en "écoutant" sa file privée de messages. La thread d'abonnement, qui écoute seule le canal, reçoit la réponse qu'elle redirige vers la file de la thread concernée.

Cette structure évite surtout à une thread applicative de monopoliser le canal de l'émission d'une demande jusqu'au retour de la réponse; dans ce cas, les communications entre le client et le serveur ne pourraient avoir un fonctionnement asynchrone déséquencé. Cette structure permet également à la thread de recevoir des messages provenant d'une des autres threads de l'Espace de Travail : c'est le cas d'une thread de service dont le client émet plusieurs demandes simultanées (i.e. client exécutant plusieurs transactions simultanées). L'Espace de Travail utilise une thread d'abonnement pour chacun de ses abonnements à des services. Dans la figure 5.12, l'Espace de Travail est abonné au service de Données du serveur X et à celui du serveur Y : deux threads d'abonnements écoutent chacune les réponses de X et Y provenant de deux canaux distincts.

Figure 5.11 : Émission de demandes et Routage des réponses.



Le routage interne des réponses est réalisé au moyen d'identifiants locaux des requêtes envoyées aux serveurs :

1- une thread émet une demande vers le serveur avec un identifiant de messages (cet identifiant est unique dans l'Espace de Travail client).

2- la réponse est retournée par le serveur avec l'identifiant de la demande. La thread d'abonnement fait alors la correspondance entre l'identifiant et la thread émettrice pour adresser à cette dernière la réponse.

Cet identifiant peut être le numéro de la thread elle-même et dans ce cas, celle-ci ne peut envoyer d'autres demandes avant d'avoir reçu la réponse de la précédente. Dans le cas des services de Données et Coopératifs[11], l'identifiant de la demande est celui de la page (i.e. celui du verrou associé) qu'une thread souhaite accéder. Ce choix d'identifiant pour ces services permet :

- à une thread de demander plusieurs pages simultanément,

  C'est le cas des threads de service dont les clients exécutent plusieurs transactions simultanées.

- à plusieurs threads de se mettre en attente d'une page déjà demandée sans émettre de messages.

  Dans l'Espace de Travail client de la figure 5.11,

  (a) les threads A et C émettent respectivement des demandes pour les pages 1 et 2. La thread B qui souhaite accéder ensuite à la page 1 n'émet pas de demande mais ajoute son nom dans la "table de correspondance" pour la réponse à la demande 1.

  (b) La réponse de la page 2 est retournée avant celle de la page 1; la thread d'abonnement route cette réponse vers la file de la thread C.

  (c) La réponse de la page 1 est routée vers les files des threads A et B conformément à cette table de correspondance. Dans le service de Données, cette table se confond avec celles de verrous.

IV.2.2. Communications Asynchrones entre le Client et Serveur

Coté serveur, l'abonnement d'un client à un service est pris en compte par une thread (d'activité) de service instanciée par l'Espace de Travail serveur au moment de la connexion du client.

Le code de la thread de service place celle-ci en attente d'un message dans sa file privée. A la réception d'un message, la thread traite celui-ci en fonction de sa nature puis se remet en attente d'un autre message. Nous illustrons la provenance des messages par la figure 5.12. Ces messages proviennent des expéditeurs suivants dans l'Espace de Travail :

- la thread d'écoute,

---

11 Rappelons qu'un service Coopératif propose les requêtes d'un service de Données, qui porte sur un même ensemble de données, étendu par les requêtes liées aux mécanismes de communication entre le superviseur central et les coopérants distribués (cf. chapitre 4 section VIII.3).

Cette thread est affectée à l'écoute des demandes (i.e. D1,D4,D5,D6) du client arrivant sur le canal de communication. Elle redirige les messages (i.e. D1,D4,D5,D6) reçus vers la file privée de la thread de service à laquelle elle est associée. L'introduction de ce type de thread dans l'Espace de Travail uniformise l'implantation des threads d'activité (applicatives et de services). Cette thread est aussi indispensable sur Solaris car dans ce système d'exploitation, une thread ne peut pas être bloquée à la fois sur la variable de synchronisation, qui constitue la file privée de message, et dans l'appel système pour l'écoute du canal.

- une thread d'activité (applicative ou de service),

  La thread de service peut recevoir des messages de la part d'une autre thread d'activité. Dans le cadre de l'accès aux données persistantes, ce message (i.e. R2) peut être l'accord d'un verrou comme dans le cas de l'Espace de Travail isolé. Ce message d'accord entraîne l'envoi d'une réponse (i.e. R2) vers le client.

- une thread d'abonnement.

  La thread de service peut appartenir à un Espace de Travail intermédiaire connecté à un ou plusieurs serveurs. Elle joue le rôle d'intermédiaire entre le client et le serveur. A ce titre, elle envoie donc des demandes au serveur comme le fait une thread applicative (i.e. la thread B envoie D4 vers le serveur Y); la réponse du serveur peut arriver après que la thread de service ait traité plusieurs messages reçus dans sa file privée; la thread d'abonnement redirige alors cette réponse vers la thread de service (i.e. la thread d'abonnement X retourne R3 à la thread A qui le renvoie vers son client A).

La file de messages évite ainsi le blocage de la thread sur l'arrivée d'un seul événement : la thread peut attendre en même temps la dépose d'un verrou ou l'arrivée d'une réponse du serveur.

Dans l'exemple de la figure 5.12, nous commentons la cause des différents messages reçus par la thread de service dans le cas d'un service de Données :

- Dans ces trois premiers cas, la page et le verrou sont disponibles sur l'Espace de Travail : la thread de service répond au client par un message d'accord du verrou avec/sans l'image de la page :

  Le message D1 du client demande le verrou et/ou l'image de la page immédiatement disponible sur l'Espace de Travail Intermédiaire.

  Le message D2 provient de la dernière thread du groupe qui possédait le verrou sur l'Espace de Travail Intermédiaire.

  Le message R3 correspond à une réponse du serveur reçue et redirigée par la thread d'abonnement X.

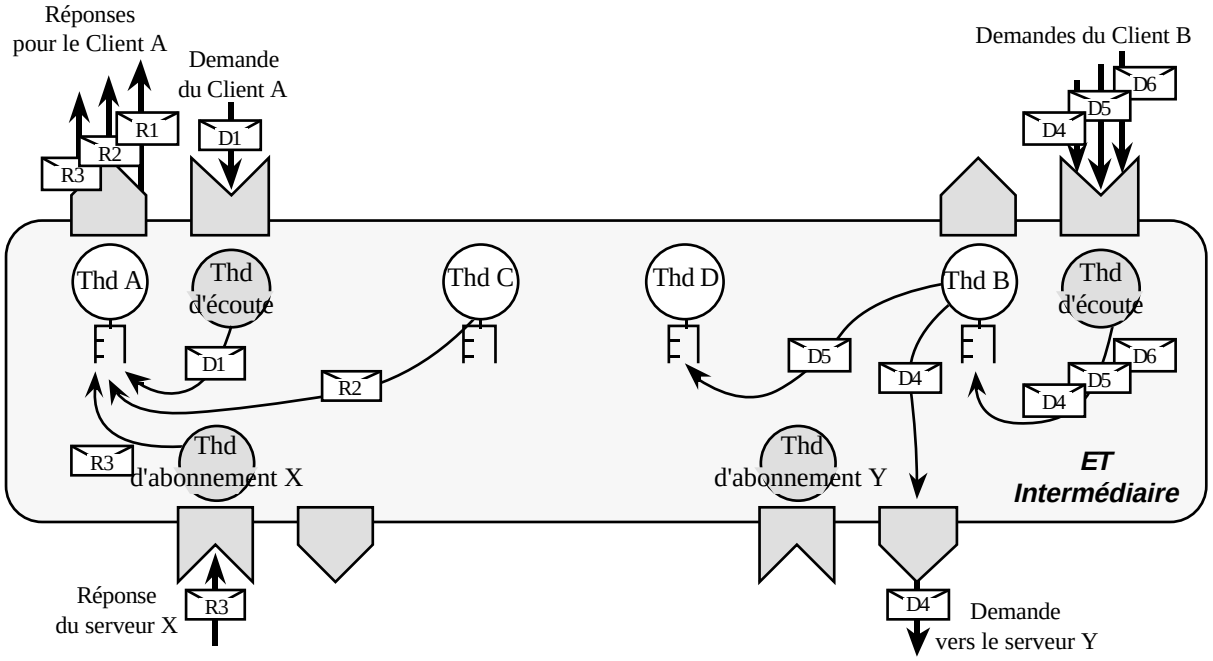
- Dans les trois cas suivants, la thread de service ne répond pas au client :

  Le message D4 correspond à la demande de page distante, qui ne peut pas être satisfaite localement, ou à la redescente d'une page modifée vers le serveur lors d'une validation.

Le message D5 correspond à la validation de page locale. Cette validation se termine par la dépose du verrou qui libère la thread D.

Le message D6 demande une page locale ou distante dont le verrou est déjà détenu par une autre thread. Dans le cas d'une page distante, le verrou a déjà été réclamée par une autre thread dans l'Espace de Travail Intermédiaire.

Figure 5.12: Transit et Traitement des messages dans un ET intermédiaire.



IV.2.3. Services Multi-Transactions.

Nous avons supposé jusqu'à présent que chaque thread d'activité de l'Espace de Travail n'exécutait qu'une seule transaction à la fois. Cette hypothèse est suffisante pour les activités applicatives ou pour des activités des services des Mixtes d'Opérations de Données (cf. chapitre 4 section VI). Cependant pour les autres services, le client exécute plusieurs transactions simultanées et donc l'activité de service, qui répond à ce service, doit instancier une transaction serveur pour chaque transaction en cours sur le client. Les services concernés sont des trois types suivants :

- le service de Données(cf. chapitre 3 section IV.2),
- les services Coopératifs (cf. chapitre 4 section IX)
- les services d'Opérations (cf. chapitre 4 section VII)

Dans ces services multi-transactions, nous distinguons deux cas :

1- le service n'a pas besoin d'un contexte de modification sur le serveur,

   C'est le cas des services de Données et les services Coopératifs qui ne contrôlent pas de contraintes d'intégrité.

2- le service requiert un contexte de modification sur le serveur.

C'est le cas des services d'Opération ainsi que ceux de Données et Coopératifs qui contrôlent des contraintes d'intégrité, au niveau du serveur, lors de la validation d'une transaction (cf. chapitre 4 section IV.3.2).

Cas de Services de Données et Coopératifs sans contrôle de contraintes d'intégrité.

Les services de Données et Coopératifs n'utilisent pas toujours la sécurisation des validations des transactions clients. Dans ce cas, les transactions serveurs n'ont pas besoin d'un contexte privé pour vérifier les données redescendues par le client.

Comme nous l'avons déjà évoqué, la thread (d'activité) de service possède la même structure que les services "mono-transaction" : dans le cadre d'une validation d'une transaction d'un client, la thread d'écoute reçoit les données modifiées par le client et redirige celles-ci vers la file privée de la thread de service. Cette dernière journalise et stocke sur le volume, les modifications reçues sans effectuer de contrôle sémantique sur les valeurs redescendues (contraintes d'intégrité).

Dans l'Espace de Travail, l'ensemble des threads des services de Données et Coopératif accède aux pages d'un segment au travers du mapping partagé de celui-ci. Ceci est permis car la thread du service n'effectue jamais l'accès implicite sur les pages; la mise en place d'un mapping privé pour détecter les violations (i.e. des accès implicites) est donc inutile. Cette optimisation évite le surcoût des mappings et permet d'économiser l'espace d'adressage de l'Espace de Travail.

Cas des Services Multi-Transactions.

Lors d'un abonnement à un service multi-transaction, chaque transaction serveur a besoin d'un espace privé de modification pour :

- modifier des données,

  Dans le cadre d'un service d'Opérations, chaque transaction serveur, associée à une transaction en cours sur le client, effectue des requêtes qui consultent et modifient les objets de l'espace de données de l'Espace de Travail serveur.

- consulter des données modifiées, redescendues par le client avant de les valider.

  Dans le cadre du service de Données (ou du service Coopératif) vérifiant les contraintes d'intégrité sur les données redescendues (cf. chapitre 4 section IV.3.2), chaque transaction serveur reçoit l'ensemble des données modifiées, de sa transaction cliente correspondante, elle vérifie ensuite les contraintes d'intégrité sur ces données avant[12] de valider la transaction.

La structure de ce type d'activité de service comporte une thread de service pour chaque transaction client et une seule thread d'écoute. Cette dernière reçoit une demande d'une des transactions du client et redirige celle-ci vers la file de la transaction serveur correspondante.

---

12 Le calcul de certaines contraintes d'intégrité ne réclame pas la totalité des données modifiées par le client. Dans ce cas, les modifications peuvent être vérifiées "à la volée" lors de la redescente et être validées simultanément. Si une de ces vérifications échoue, la validation doit être abandonnée et défaite par le mécanisme de reprise sur panne (section IV.3). Si cet événement est rare, cette technique peut être envisageable. Si les contraintes sont toutes de ce type, la thread de service n'a plus besoin de mapping privé.

L'identifiant de la demande comporte en autre le numéro de la transaction client pour permettre le routage des demandes par la thread d'écoute.

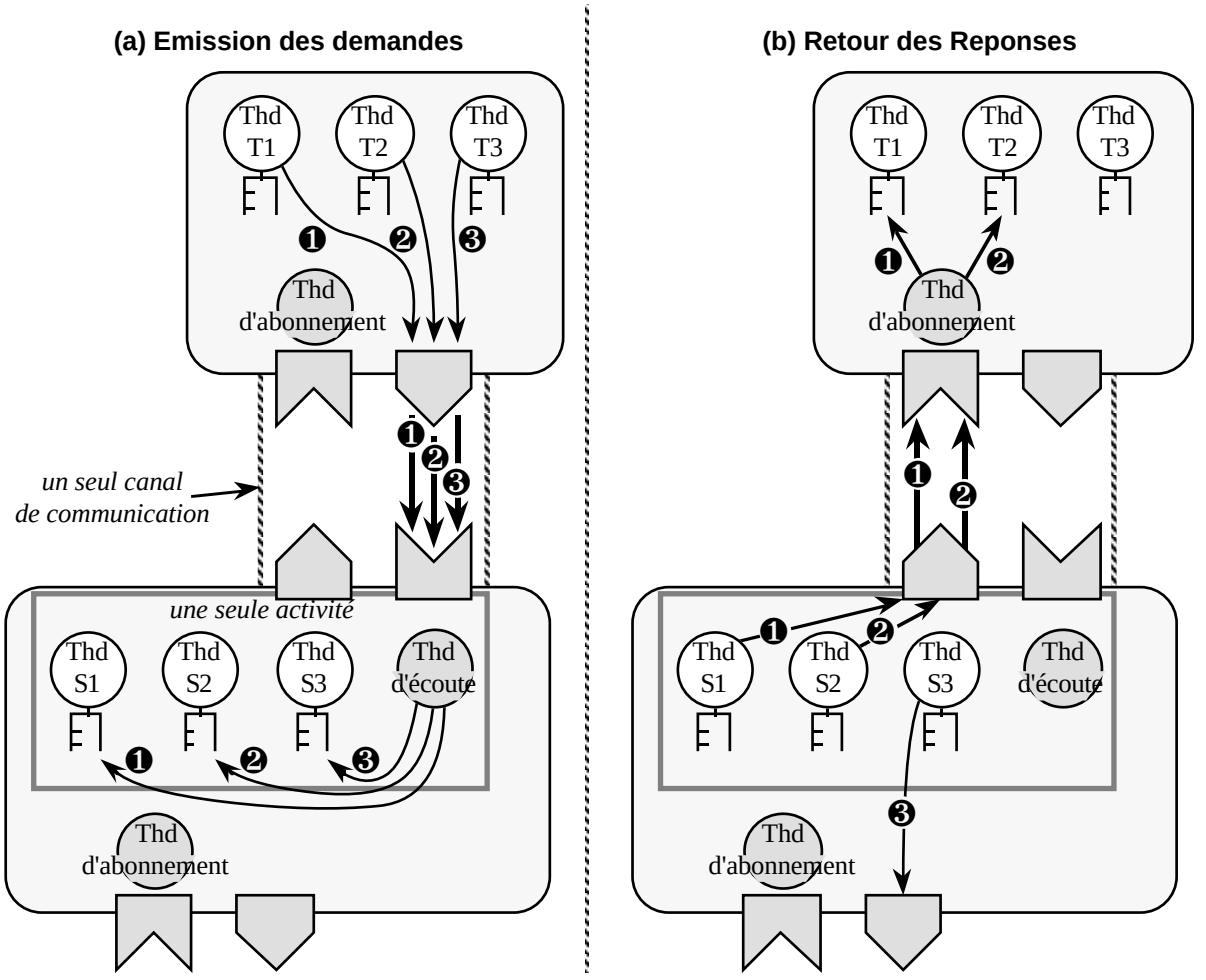
Dans la figure 5.13, l'activité de service comporte autant de threads que de transactions en cours sur le client.

(a) Les transactions T1, T2 et T3 du client envoient des demandes (de calcul ou de page selon le service de l'abonnement) vers le client. Ces demandes sont routées vers les threads de service respectives.

(b) Dans le cas présent, les threads S1 et S2 répondent au client alors que la thread S3 propage la demande de T3 vers un Espace de Travail serveur. Les threads de service agissent indépendamment les unes des autres même si elles se partagent le même canal de communication pour les réponses et la même thread d'écoute pour la réception.

Chaque thread de service utilise un mapping privé pour réaliser son espace privé de modification. Cependant, le cas du service de Données (Coopératif), la thread utilise le mapping partagé commun de l'Espace de Travail jusqu'au moment de la validation; elle s'alloue un

Figure 5.13 : Prise en compte d'un abonnement par une activité multi-transaction.



mapping privé seulement au moment de la vérification. Cette allocation retardée évite l'allocation inutile d'un mapping privée en cas de transaction uniquement consultative ou en cas d'abandon de la transaction; d'autre part, la présence du mapping privé dans l'espace d'adressage ne dure que le temps de la vérification.

IV.3. Journalisation Parallèle

IV.3.1. Rappel sur la Journalisation

Les pages modifiées sont écrites dans un journal avant d'être écrites dans la base. Le journal sert à récupérer une image stable de la base contenant les modifications apportées par toutes les transactions validées.

- une validation incomplète de la transaction,

  La transaction entame sa validation mais ne la termine pas (en Client-Serveur, ce cas peut correspondre à l'arrêt du client ou à la rupture de la connexion) : l'image de la base contient des modifications de la transaction qui ont été écrites dans la base. Le journal sert à retrouver les anciennes valeurs de la page pour remplacer les modifications effectuées (undo). Cette opération est effectuée sans arrêter le travail des autres transactions du système (online).

- un arrêt accidentel du système,

  L'image de la base contient des modifications de transactions arrêtées en cours de validation (cas précédent) mais également des transactions validées dont les pages n'ont pas été encore écrites dans la base. Dans ce dernier cas, les modifications de ces dernières transactions sont extraites du journal pour être appliquées à la base (redo).

- après une panne de média,

  L'image de la base est physiquement endommagée : elle est reconstruite à partir d'une sauvegarde de la base (checkpoint) sur laquelle sont appliquées les modifications journalisées par les transactions qui ont terminé leur validation avant la panne.

IV.3.2. Structure de l'Organe de Journalisation

L'opération de journalisation correspond à une écriture conventionnelle (i.e. appel système `write()`) dans un média séquentiel (fichier, disque ou bande). Afin d'améliorer le débit d'écriture, le journal est constitué de plusieurs médias physiquement distincts. Cette démarche permet de paralléliser les écritures entre les différents médias. Les choix de conception de l'organe de journalisation sont :

- d'utiliser le MultiThreading pour paralléliser les écritures sur les journaux et pour paralléliser les opérations de diffing,
- de favoriser le remplissage des blocs d'écriture vers les journaux,
- de réduire les déplacements de zones mémoires à l'intérieur de l'Espace de Travail.

L'organe de journalisation reçoit de la part des threads d'activités, 3 types de données à écrire sur les journaux:

T- un buffer totalement rempli d'enregistrements de journalisation,

ce buffer provient sans doute d'une thread de service[13] qui valide les modifications de son client; ce buffer contient en fait le message du client. Le client a rempli ce message avec les enregistrements de journalisation à redescendre. Le format utilisé pour ce message est compatible avec celui des blocs du journal pour éviter les conversions et les copies mémoire sur le serveur. Ce message reçu dans le buffer peut être directement écrit sur un des journaux.

P- un buffer partiellement rempli d'enregistrements de journalisation,

ce buffer contient le dernier message de journalisation envoyé par le client. Ce message n'occupe pas la totalité du buffer; l'organe de journalisation doit tenter de compléter ce buffer avant de l'écrire dans le journal.

D- une page modifiée à journaliser.

les modifications apportées à cette page doivent être journalisées. Pour cela, nous utilisons la technique de "diffing" qui consiste à ne journaliser que les différences entre l'image modifiée et l'image originale de la page. Cette opération, qui est détaillée dans la section V.4, utilise ces différences pour constituer les enregistrements de journalisation. L'opération de diffing crée directement ces enregistrements, de préférence dans les buffers partiellement remplis de la catégorie précédente (P), et sinon dans un buffer interne de l'organe de journalisation

L'organe reçoit, de la part des threads d'activité, des demandes de journalisation pour ces trois catégories; les messages sont postés dans 3 files T(otal), P(artiel), D(iffing) correspondants aux 3 catégories précédentes. L'organe de journalisation se compose d'un pool de threads écrivains et d'un pool de threads de diffing :

- Chaque thread du pool des écrivains pilote les écritures vers un journal attitré. Chacune s'occupe d'écrire le contenu d'un buffer provenant de la file T(otal) vers son journal. Une fois l'écriture terminée, la thread écrivain notifie l'écriture du buffer aux threads d'activité qui s'étaient placées dans la liste des notifications de l'écriture. Les notifications sont des messages envoyés dans les files de message des threads d'activité.
- Chaque thread de diffing s'occupe de remplir les buffers de la file P(artiel) avec les enregistrements générés par le diffing d'une page retirée de la file D(iffing). Les enregistrements de diffing d'une page peuvent être regroupés dans le même buffer[14] ou bien être répartis sur plusieurs. Une fois qu'un buffer de P est rempli en totalité, une demande de journalisation est glissée dans la file T pour écrire le buffer rempli sur un des journaux avec une liste de notifications. Cette liste contient l'identité de la thread à qui appartient le buffer, ainsi que l'identité des threads qui ont un enregistrement de diffing dans ce buffer. Cependant pour ces dernières, la notification est conditionnée

13 Service de Données pour un client en mode Passant.

14 En section V.4.3, nous verrons que ces enregistrements n'en forment plus qu'un seul dans un buffer, afin d'éviter le surcoût des entêtes.

par le fait que ce buffer soit le dernier qui contienne un des enregistrements de diffing de la page[15] .

Si la file D(iffing) ne contient pas de demande de diffing, une thread de diffing fusionne les buffers de la file P ensemble pour remplir certains d'entre eux; elle demande ensuite l'écriture de ceux-ci en glissant les demandes dans la file T.

Si la file P(artiel) ne contient pas de buffers partiellement remplis et qu'il existe des demandes de diffing dans la file D, le pool de threads de diffing allouent un buffer, interne au pool de diffing, pour y réaliser les diffing en attente. Si le buffer est complet, il est basculé vers la file T(otal) pour être écrit. Si par contre, il n'est que partiellement rempli, il fait un "court" séjour dans la file P pour y attendre une demande de diffing ou un buffer P avec qui fusionner. Si le séjour est trop "long", le buffer même partiellement rempli est basculé vers la file T.

Les threads de diffing se partagent le diffing des pages en attente dans la file D. Chacune de ces threads s'approprie un buffer de la file P ou un buffer interne. Pour permettre que les différents buffers de la file P se remplissent rapidement, le nombre de threads actives dans le pool de diffing est adapté en fonction du taux de fusion des buffers de P et en fonction du taux des buffers de P basculés dans T alors que ceux-ci n'étaient pas remplis. D'autre part, une thread de diffing ne s'alloue pas de buffer interne tant qu'il existe des buffers de la file P en cours de remplissage. Ces différentes règles permettent d'adapter le travail du pool de diffing au débit d'écriture vers les journaux.

Dans l'exemple de la figure 5.14, une thread de service Ts demande l'écriture des 2 buffers B1, B2 vers le pool de journaux J0 et J1. Un troisième buffer B3 de Ts n'est rempli que partiellement; Ts formule une demande de remplissage dans la file P(artiel). Simultanément, une thread applicative Ta dépose des demandes de journalisation pour les 3 pages P1, P2 et P3 dans la file D(iffing). L'organe de journalisation effectue en parallèle l'écriture des buffers et ainsi que le diffing des pages qui rempli les buffers partiellement remplis de la manière suivante :

a- Les buffers B1 et B2 sont écrits en parallèle sur les journaux J0 et J1 par les threads écrivains E0 et E1; une fois qu'une écriture est effective sur le média, chaque thread écrivain notifie à la thread à qui appartient le buffer la terminaison de l'écriture sur le buffer. Ts qui reçoit ces notifications pour les buffers B1 et B2 peut réallouer ceux-ci à autre chose (autre réception par exemple).

Pendant le blocage des threads E0 et E1, une thread de diffing D0 "diffe" la page P1 et une partie de la page P2 (partie modifiée b) pour compléter le buffer B3.

b- Une fois le buffer B3 complété, une demande d'écriture de B3 est déposée dans la File T. Une des deux threads écrivains, E0, écrit le buffer B3 dans le journal J0, et notifie ensuite à la thread Ts de l'écriture de son buffer B3, et à la thread Ta de l'écriture de

---

15 La condition est donnée pour un compteur, associé à la page, que décompte le nombre de buffers restant à valider avant de notifier la thread d'activité de l'écriture de tous les enregistrements de diffing. Il est initialement fixé à une très grande valeur car le diffing ne sait pas d'avance combien de buffers vont être remplis des enregistrements de diffing de la page. Chaque fois qu'une thread écrivain écrit un buffer contenant un de ces enregistrements, elle décrémente ce compteur de un. La thread de diffing rajuste le compteur quand elle a terminé le diffing en comptant les décrémentations déjà effectuées. Quand le compteur passe à zéro, la thread écrivain peut notifier la thread.

tous les enregistrements de P1. Cette dernière peut alors écrire la page modifiée P1 sur l'archive (cf. section III.3.3). En revanche, Ta ne reçoit pas de notification pour P2 car conformément à la règle du Commit, la page P2 ne doit pas être écrite sur l'archive tant que tous les enregistrements de journalisation, qui la concernent, n'ont pas été écrits.

Pendant le blocage de la thread E0, la thread de diffing D0 "diffe" la page P3 et le restant de la page P2 sur un buffer interne Bi car la file P ne contient plus de buffers à compléter.

c- Une fois le buffer Bi complété, une demande d'écriture de Bi est déposée dans la File T. Une des deux threads écrivains, E1 (puisque l'accès à la file est alterné), écrit le buffer Bi dans le journal J0. Elle notifie ensuite la thread Ta de l'écriture de tous les enregistrements de P2 et P3. Elle libère aussi le buffer interne.

La structure de l'organe de journalisation exploite plusieurs formes de parallélisme :

- Parallélisme des écritures sur les différents journaux.

  Il faut pour cela que la machine soit équipée de plusieurs médias physiques pilotés par des contrôleurs indépendants.

- Parallélisme entre les écritures et le diffing des pages.

  Sur une machine monoprocesseur, l'écriture vers le journal est bloquante pour la thread qui pilote le journal; la thread de diffing utilise le processeur pendant ce blocage pour "differ" les pages demandées dans la file D.

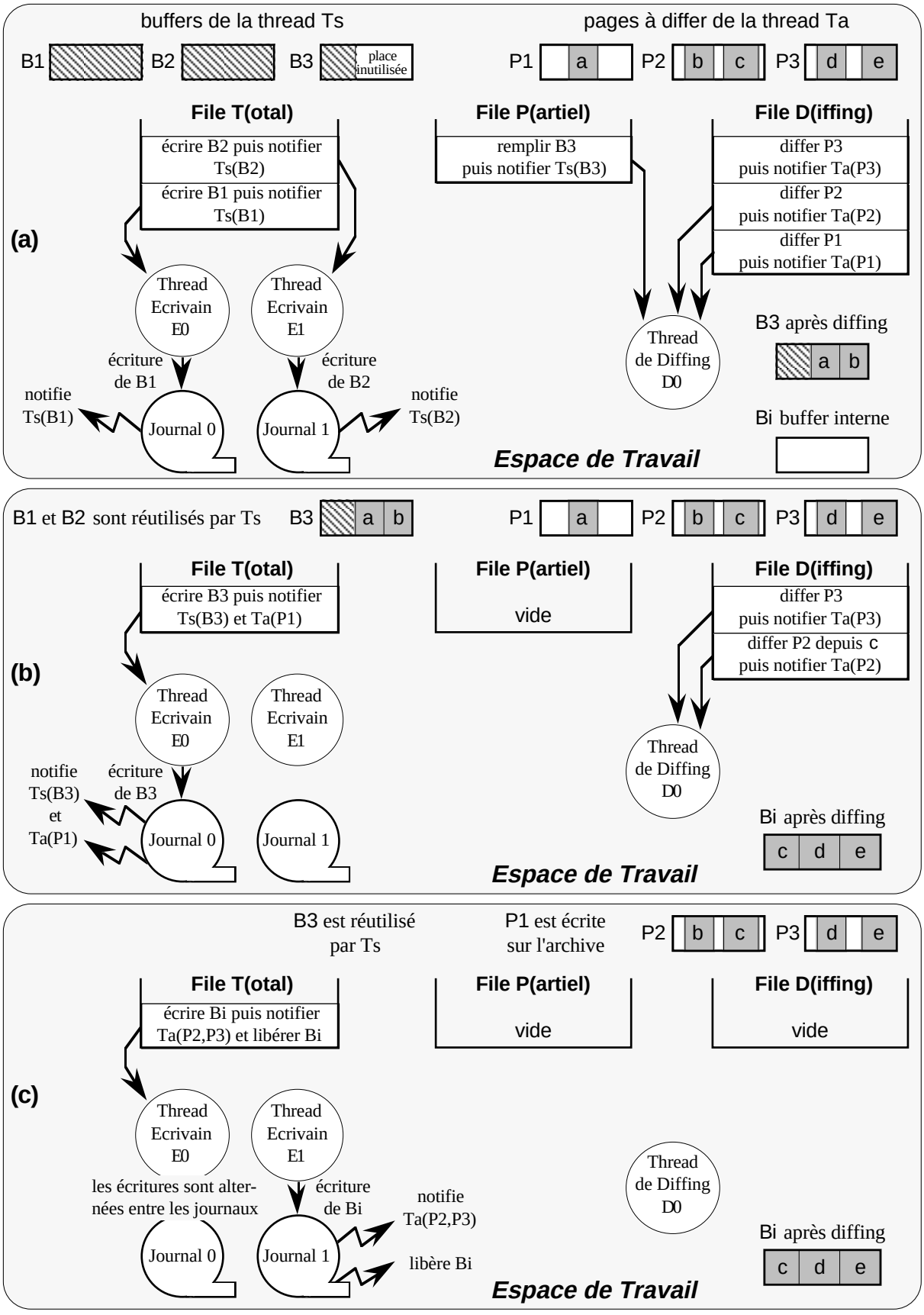
- Parallélisme dans le diffing des pages.

  Sur une machine multiprocesseur, plusieurs threads de diffing peuvent "differ" des pages indépendantes vers des buffers indépendants. Les pages modifiées d'une thread d'activité sont ainsi diffées en parallèle si la thread dépose plusieurs demandes dans la file D sans attendre les notifications de fin d'écriture.

En revanche, l'opération de reprise sur panne n'est pas parallèle : les journaux sont remontés en suivant un numéro global (à l'organe) d'enregistrement. Nous étudions la possibilité de paralléliser cette opération en exploitant la présence des enregistrements de journalisation d'une thread dans les buffers de relecture des journaux.

Notre organe permet de faire côtoyer la journalisation de threads de services de Données (principalement des buffers déjà remplis) avec celle des threads applicatives ou des autres services (réalisation de diffing) en favorisant le remplissage des buffers écrits sur les journaux.

Figure 5.14 : Journalisation parallèle dans un Espace de Travail



V. Accès aux Objets

La page est le grain choisi pour manipuler les données sur un Espace de Travail qu'il soit client ou serveur. L'interface d'accès aux pages ne considère pas le contenu des pages. La couche Objet de WEA définit une interface d'accès aux objets. Cette couche structure le contenu des pages avec la notion d'objets. Il existe les deux types d'objets: les objets courts et les objets longs. Les objets courts ont une taille inférieure à celle de la page physique et une page regroupe plusieurs objets courts. Les objets longs sont des objets multi-pages dont les pages sont installées de façon contiguë dans l'espace d'adressage de l'Espace de Travail. Les objets courts peuvent d'être instanciés de façon persistante ou de façon temporaire : un objet instancié temporaire disparaît en fin de transaction alors que l'objet instancié persistant est créé dans un volume local ou distant de la base lors de la validation.

La première section décrit la structuration des pages en objets courts et longs. La section suivante détaille l'instanciation temporaire ou persistante des objets. La section V.3 présente les identifiants et le mécanisme de déréférenciation. Enfin nous terminons avec l'amélioration de la journalisation en tenant compte de la structure des pages.

V.1. Objet Court / Objet Long

La couche d'accès aux objets structure le contenu des pages en objets non significatifs. Cette couche n'interprète pas le contenu de ces objets qui est du ressort de la couche langage. La couche Objet propose deux types d'objets qui se distinguent par la nature de leur stockage :

- les Objets Courts,
- les Objets Longs.

V.1.1. Objet Court

La taille d'un objet court est toujours inférieure à celle d'une page manipulée par l'Espace de Travail. Plusieurs objets courts sont regroupés dans une page physique. L'organisation de la page suit celle des "slotted pages" : la page est découpée en slots égaux (tranches) et les objets courts sont placés sur un nombre entier de slots (figure 5.15). Les slots en entête de la page sont réservés à une table d'objets.Cette table associe un numéro d'objet relatif dans la page au numéro du slot contenant le début de l'objet. Un choix judicieux de la taille du slot permet de rendre compacte chaque entrée dans la table[16]. La table constitue la dernière indirection pour accéder aux objets une fois que l'adresse de la page a été calculée (§V.3.2). L'entête de l'objet contient la longueur utile de l'objet et d'autres informations liées à la persistance ou utilisées par la couche Langage (comme le type des instances générées par cette couche).

---

16 Une page de 4Ko contient 256 slots de 16 octets : le numéro de slot est codé sur 1 octet. Comme les premiers slots sont occupés par la table d'objets, leurs numéros sont utilisés pour coder la destruction d'un objet ou un numéro libre d'objet.

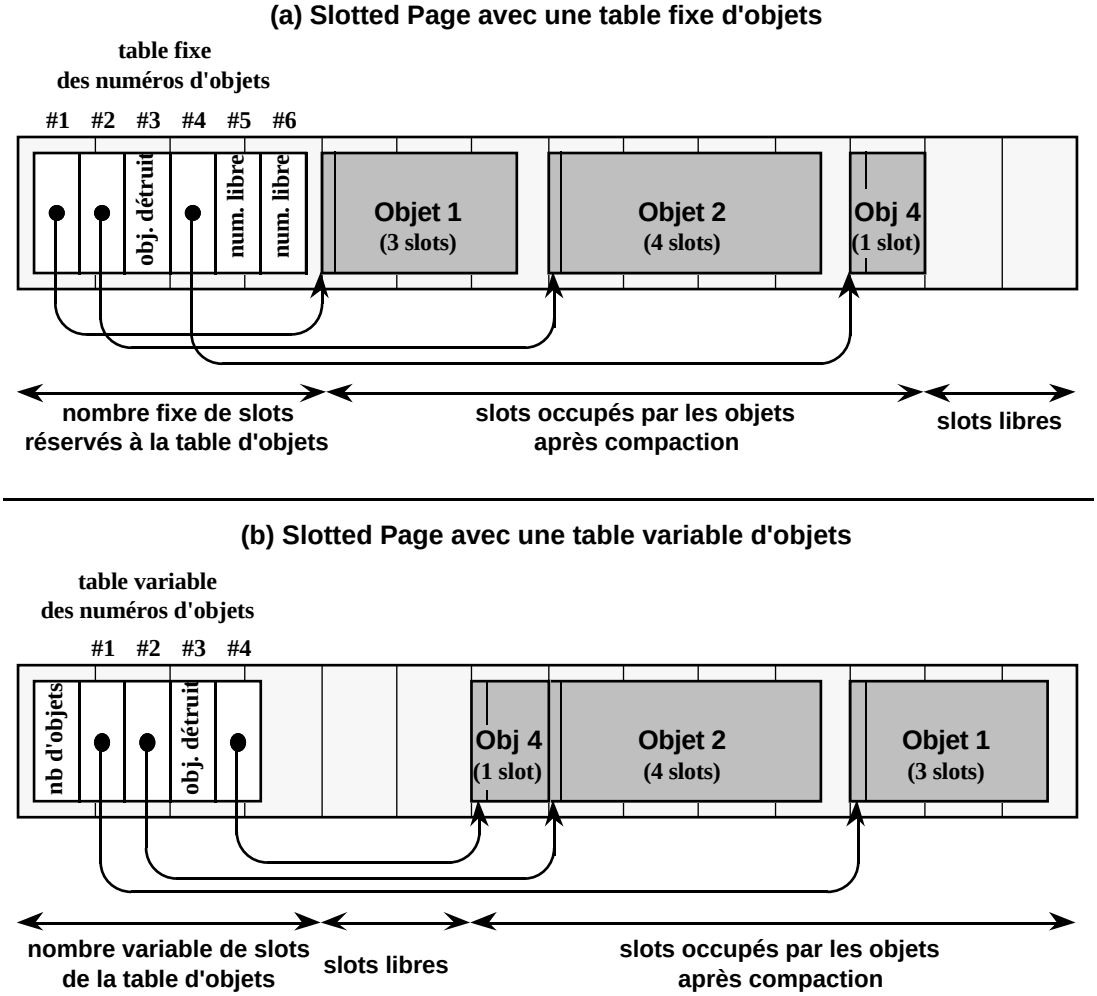
L'indirection sur les objets dans les slotted pages permet de réaliser une compaction plus simple des pages d'objets courts quand un ou plusieurs objets ont été détruits dans la page.Le numéro des objets détruits est conservé dans cette table; ceci permet de détecter les "pointeurs pendants" (Dangling Pointers") référençant des objets détruits.

Nous proposons deux types de structure pour les slotted pages :

(a) La première structure comporte en entête une table d'objets de taille fixe. Cette table contient le numéro du slot qui contient le début de l'objet court pour chaque numéro d'objet défini dans la page. Des valeurs spéciales sont utilisées pour marquer les entrées des objets détruits (quand les numéros de ces objets ne doivent pas être réalloués) ou les entrées des objets qui n'existent pas encore.

(b) La seconde structure comporte en entête une table d'objets comportant un nombre variable d'entrées. Comme dans la structure précédente, cette table met en correspondance un numéro d'objet dans la page avec le numéro du slot contenant le début de l'objet court ou l'indication que l'objet a été détruit. Cependant, la table ne réserve pas de slots à l'avance pour stocker les entrées des numéros d'objets qui n'existent pas encore. La table grandit au fur et à mesure des besoins en identifiants

Figure 5.15 : Structuration du contenu de la page d'objets courts.



d'objets (cette opération n'est réalisable que s'il reste un slot libre consécutif à la table). La table peut aussi réduire sa taille si les entrées en fin de table sont libérées. Cette taille dynamique rend cette structure préférable à la précédente notamment avec l'utilisation d'identifiants longs (64 bits ou plus) : la table peut contenir jusqu'à 4000 entrées[17] pour une page de 4Ko dans une page vide d'objets "vivants".

V.1.2. Objet Long

L'objet long est un objet multi-pages : c'est un long tableau d'octets chargés dans un ensemble de pages contiguës dans l'espace d'adressage de l'Espace de Travail.

L'objet long est aussi connu sous le nom de BLOB (Binary Long OBject) et il est utilisé pour archiver des données multimédia telles que des images ou des sons. La contiguïté des pages en mémoire virtuelle assure la manipulation de ces objets par des accélérateurs matériels [Hennessy90] ou l'utilisation par l'Espace de Travail de bibliothèques (librairies) externes de fonctions pour manipuler les données archivées à l'intérieur des objets.

En revanche, la contiguïté des pages sur le support stable ne peut être garantit que si le volume WEA est implanté directement avec un disque "raw" sans passer par l'organisation du système de fichiers. La contiguïté des pages d'un objet long sur disque est particulièrement recommendée dans le cas de l'archivage de données multimédia : le chargement de ces données en mémoire est en général soumi à des contraintes temps réel. Les systèmes de fichiers classiques ne supportent pas ces contraintes; le multimédia requiert une organisation physiquement séquentielle pour ce type d'objets afin de réaliser des échanges pour des blocs de plus grandes tailles [Reddy94]. Nous verrons dans la section V que cette hypothèse de contiguïté disque est utilisée pour implanter la classe Large qui instancie de longues chaînes de caractères.

L'extension d'un objet long est une opération hasardeuse. Elle ne peut s'effectuer que si les pages qui suivent celles de l'objet sont libres et que si l'ajout reste dans les limites du segment dont la taille a été fixée à la création.

Pour simplifier la gestion des pages dans les segments, les objets longs et les objets courts sont instanciés dans des segments séparés : les segments d'objets courts et les segments d'objets longs. La couche Page ne distingue pas ces deux types de segment; les pages sont ainsi accédées dans les deux cas par mapping du segment entier. Cette approche simplifie beaucoup la conception de l'Espace de Travail mais elle ne prend pas en compte les spécificités d'accès des objets longs. En général, les pages d'un objet long sont accédées séquentiellement : la couche Page peut détecter les demandes séquentielles dans un objet long et, dans un contexte client-serveur, le client peut ainsi importer le reste de l'objet long avec une seule demande.

V.2. Instanciations Persistantes et Temporaires.

Un objet court est instancié soit de façon permanente (persistante), soit de façon temporaire. Un objet créé permanent est réellement alloué dans la base lors de la validation de la transaction alors qu'au même moment, l'objet créé temporaire disparaît. Les objets temporaires n'existent que le temps de la transaction. Néanmoins, une méthode `setPersistent()` accessible au

17 Avec des slots de 16 octets dans une page de 4Ko.

développeur permet de rendre permanent un objet créé temporaire et de "survivre" à la terminaison de la transaction.

Les objets temporaires sont alloués dans le tas (heap) privé de la thread : ce tas privé permet une désallocation rapide des objets temporaires à la terminaison d'une transaction. Ce tas est géré sans souci de frontière de page et avec des slots égaux en taille à ceux des pages persistantes. L'identifiant d'un objet temporaire est donc une adresse dans l'espace d'adressage valide jusqu'à la terminaison de la transaction. L'objet temporaire reste dans ce tas même s'il a été rendu persistant par la méthode `setPersistent()`. La validation de la thread crée de nouveaux objets persistants pour y copier l'image des objets temporaires devenus persistants. Les objets temporaires désormais persistants se voient attribués ainsi un nouvel identifiant dans des pages persistantes. Nous verrons que la couche Langage remet à jour les références, qui peuvent exister vers ces objets, dans les objets nouvellement créés ou modifiés par la thread au moment de la validation.

Les objets courts persistants sont directement alloués dans des pages choisies dans l'ordre de priorité suivant :

1- L'objet est prioritairement créé dans une page déjà verrouillée pour modification.

2- Si aucune page modifiée n'a assez de place libre (slot) pour y stocker l'objet, la transaction choisit une page consultée parmi celle d'un segment mappé par la thread puis demande le verrou exclusif correspondant.

3- Si aucune page consultée ne convient, la transaction demande une page nouvelle dans un segment mappé à l'intérieur de l'Espace de Travail. Cette demande peut entraîner la création d'un nouveau segment dans un volume.

4- Enfin la transaction s'alloue une page nouvelle. Cette demande peut entraîner la création d'un nouveau segment dans un volume.

La création d'un objet à proximité (nearby) d'un objet existant correspond au placement des deux objets dans la même page (clustering) : ce placement des objets à la création optimise les accès disque et réseau quand ces deux objets sont utilisés simultanément lors d'une transaction. La création de proximité impose la demande du verrou en écriture sur la page par la transaction. Cette demande n'entraîne pas forcement un surcoût de verrouillage: l'objet créé est souvent référencé par l'objet existant qui modifie son contenu pour prendre en compte la nouvelle référence ce qui nécessite de toute façon d'obtenir le verrou en écriture. Enfin si la création de proximité échoue dans la page, l'objet est placé si possible dans une page du même segment.

Dans l'environnement Client-Serveur, le client maintient un buffer de pages nouvelles (une liste de numéros de pages libres) pour les segments déjà chargés par le client pour les allocations d'objets courts. Quand le client recharge ce buffer, le serveur lui envoie un quota de numéro de pages libres. Ce quota varie dans le temps en fonction de la fréquence des demandes. Ce buffer évite au client de faire des demandes fréquentes pour un seul numéro de page. Dans le cas des objets longs, les demandes sont traitées au coup par coup sans bufferisation car les contraintes de contiguïté obligent le serveur à trouver un espace libre dans les segments déjà chargés pour y stocker l'objet long.

V.3. Identifiants d'Objet et Déréférenciation

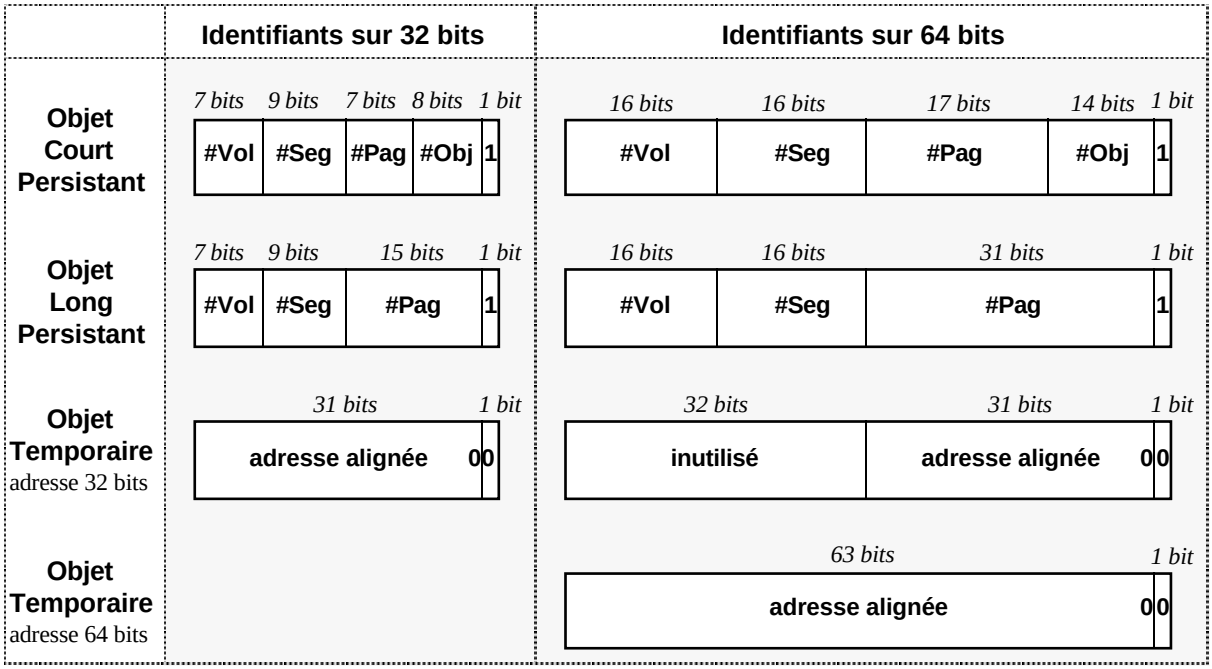
V.3.1. Identifiants d'Objet

Un objet est accessible par son identifiant. L'identifiant d'objet est une référence unique dans l'ensemble du système [Khoshafian86]. Quand l'objet désigné est détruit, sa référence n'est pas réallouée à un autre objet. L'absence de réallocation résout le problème des pointeurs pendants ("Dangling pointers") qui référencent des objets détruits. Une référence vers un objet détruit retourne une valeur nulle. Toutefois, il est possible d'échapper à cette règle de non-réallocation des identifiants notamment dans l'allocation d'objets composants un objet composite comme ceux fournit par la bibliothèque de la couche Langage (Listes, Ensembles, String, ...).

Le format d'un identifiant d'objet diffère suivant le mode d'instanciation (persistant ou temporaire) et également suivant le type de l'objet (objet court ou objet long) :

- Comme nous l'avons déjà évoqué dans la section précédente, l'identifiant d'un objet temporaire est une adresse directe dans l'espace d'adressage.
- L'identifiant d'un objet court persistant est constitué d'un numéro de volume (ce qui sert à localiser l'Espace de Travail qui le sert), d'un numéro de segment dans le volume, d'un numéro de page relative au début du segment et d'un numéro d'objet dans la page.
- Comme l'objet long occupe un nombre entier de pages dans un segment, l'identifiant d'un objet long persistant est composé des numéros du volume, du segment et du numéro de la première page où l'objet long commence. L'identifiant d'un objet long échappe à la règle de non-réallocation des identifiants.

Figure 5.16. : Structure des identifiants d'objets en codage 32 et 64 bits



La taille de l'identifiant est un des paramètres Instanciations du système: l'identifiant peut être codé sur 32 bits ou sur 64 bits. Le choix doit être fait par rapport à la quantité d'objets qui peuvent être créés durant toute la vie de cette base. Le codage sur 32 bits offre au maximum une base de 32 Go d'objets courts[18] avec un maximum de 4To d'objets créés dans la vie de la base (pour une base vide d'objets longs). La base d'objets longs contient un maximum de 4 To[19] quand la base est vide d'objets courts. Le codage sur 64 bits offre une taille de base quasi-illimitée mais augmente la taille des références persistantes dans les objets. Certains concepteurs de Gérants d'Objets préconisent des identifiants plus larges sur 128 bits [Carey94] considérant que l'espace d'identifiants sur 64 bits s'épuise très vite dans le cadre d'une base de grande échelle [Singhal92] malgré l'impression d'infinité qui a été avancé par [Chase92]. La figure 5.16. représente la structure des identifiants d'objets suivant les codages sur 32 et 64 bits. Dans les deux cas, l'identifiant temporaire et l'identifiant persistant se distinguent par leur bit de poids faible[20]. Les identifiants d'objets longs ou d'objets courts sont distingués au moment de la déréférenciation.

L'identifiant d'objet est en fait un identifiant physique qui enregistre la position de l'objet par rapport au disque. Cet identifiant ne permet pas de regrouper physiquement les objets après l'instanciation de ceux-ci. Ce regroupement physique est possible si la couche Langage crée un niveau supplémentaire d'identification pour représenter les références d'objets.

V.3.2. Déréférenciation

La déréférenciation de cet identifiant est l'opération de traduction de l'identifiant d'un objet vers l'adresse mémoire où est installée une image de cet objet. Dans le cas de l'identifiant physique, la traduction consiste :

1- à obtenir l'adresse de début du mapping privé de ce segment à partir de la table des segments mappés par la thread (figure 5.17.a).

   La table privée est une table de hachage qui ne contient les segments mappés par la thread. Si la table ne contient pas d'entrée pour le segment [#V,#S], la thread doit donc consulter des tables de volumes locaux ou distants de l'Espace de Travail pour pouvoir mapper le segment voulu. Ces tables de volumes sont partagées entre les threads de l'Espace de Travail.

2- l'adresse de la page est calculée comme un déplacement relatif par rapport au début du mapping privé de ce segment (figure 5.17.b).

3- la traduction se termine par l'indirection de table d'objets en entête de la page qui donne la position du premier slot de l'objet (figure 5.17.c).

   C'est ce premier accès dans la page qui provoque le verrouillage de la page (qui est demandé implicitement en lecture).

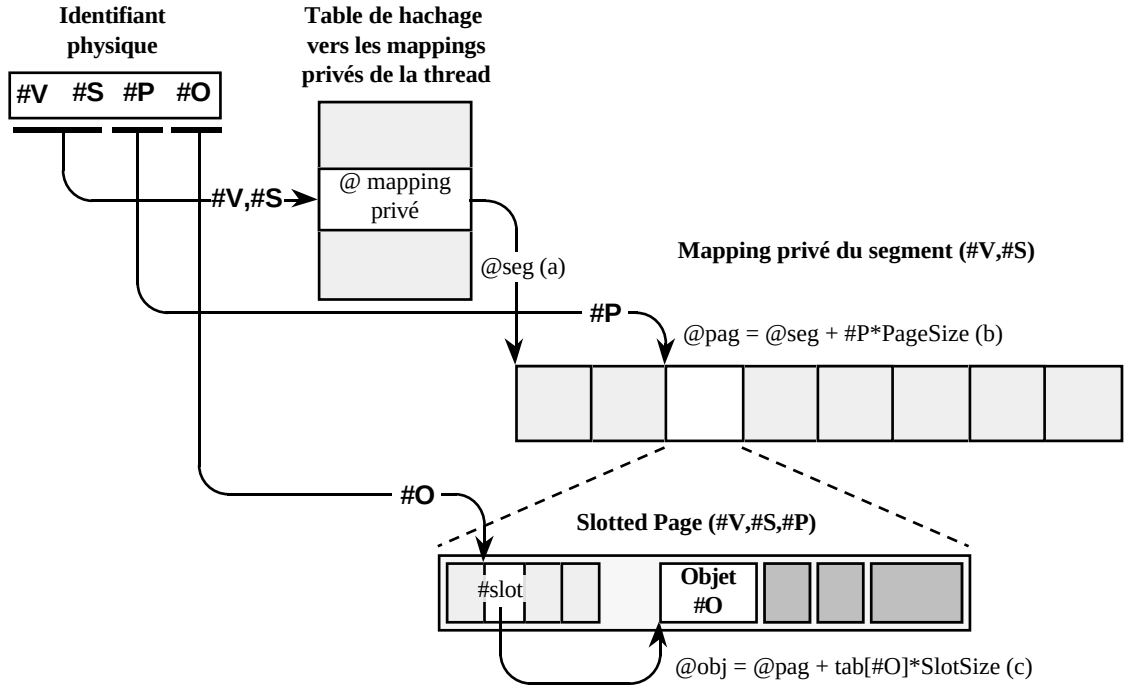
---

18 La structure de l'identifiant, codé sur 32 bits (figure 5.ID), donne un maximum de 2^{23} pages de 4Ko (soit 32Go) avec un maximum de 2^{31} objets d'au maximum 4Ko créés durant la vie de la base (soit 4To).

19 L'identifiant d'objet long code un maximum de 2^{31} pages de 4Ko réservés aux objets longs.

20 Le choix du bit de poids faible part de la remarque que les processeurs 32 bits fonctionnent de manière optimale (68040,80486, Pentium, PPC601, HP-PA) ou correctement (Sparc) pour des structures alignées sur des frontières de mots de 32 bits : les 2 bits de poids faibles d'adresses de structure ont la valeur constante 00.

Figure 5.17 : Déréférenciation d'un identifiant physique



V.4. Journalisation des Modifications sur les Objets

Dans la section III.3.3, la couche Page journalise les pages modifiées avant de les écrire dans les volumes pour refaire les modifications en cas de pannes. L'enregistrement de journalisation contient les différences entre la page initiale et la page modifiée. Cette technique dite de **diffing** permet de réduire la quantité de données à journaliser [Singhal92]. En revanche en cas de destruction du stockage (media crash), la recontruction d'une page nécessite de repartir d'une image journalisée de la page (checkpoint) et d'y appliquer les modifications journalisées (diffing).

Dans un contexte Client/Serveur, l'Espace de Travail client ne redescend jamais l'image complète d'une page modifiée vers le serveur mais il lui envoie l'enregistrement de journalisation. Le serveur écrit directement l'enregistrement dans le journal et recopie dans l'image locale de la page les différences contenues dans l'enregistrement (redo-at-server). La technique de diffing permet dans ce cas de réduire le coût des communications à la "redescente" pour un coût de calcul qui reste très faible par rapport au coût de la réception d'un message.

Nous utilisons deux techniques de diffing pour construire l'enregistrement de journalisation suivant la nature de la page journalisée :

- le Page Diffing pour les pages d'objets longs,
- l'Object Diffing pour les pages d'objets courts.

V.4.1. Page Diffing

Le page diffing est une technique classiquement utilisée dans les Gérants d'Objets [Singhal92, Carey94]. Le page diffing compare l'image modifiée d'une page avec son image

originale pour générer l'enregistrement de journalisation de cette page. Cette opération est limitée par le débit de la mémoire vers le processeur (memory-intensive)[21] et elle entraîne un faible surcoût par rapport à la copie mémoire qui est nécessaire pour journaliser une page sans utilisation du diffing[22].

L'inconvénient des techniques de diffing (page ou objet) est qu'il est nécessaire d'avoir l'image originale de la page chargée en mémoire primaire. Or après la duplication de l'image par copy-on-write, l'image originale peut être inutilisée pendant jusqu'à la validation; cette situation provoque rapidement l'évincement de l'image originale de la mémoire primaire. Les concepteurs de Texas [Singhal92] proposent des solutions pour éviter cette situation en journalisant par diffing les pages en cours de transaction. Malheureusement dans l'implantation des accès par Memory-Mapping, la page originale est rechargée en mémoire primaire quelque soit la technique de journalisation choisie (journalisation entière de la page ou celle par diffing). Dans un contexte client/serveur, le surcoût du chargement est compensé par le gain réalisé sur les communications.

Dans le page diffing, la comparaison des deux pages n'interprète pas le contenu des images des pages. C'est dans cette optique que nous utilisons cette technique pour comparer les pages modifiées dans les objets longs. Pour un objet long, le diffing est réalisé indépendamment pour chaque page modifiée et il génère donc un enregistrement de journalisation par page même si les modifications dans l'objet se trouvent à des frontières de pages.

V.4.2. Object Diffing

L'object diffing prend en compte l'existence d'un entête dans la page et de sa structure en slot lors de la comparaison des deux images. Cette technique détecte ainsi la nature des modifications opérées sur les objets de la page (création, modification d'une partie de l'objet, augmentation ou diminution de sa taille, destruction) et enregistre cette nature avec les modifications détectées dans les objets.

L'object diffing n'améliore que modérément les performances du page diffing quand la page n'est pas complètement occupée par les objets (car la comparaison ne porte pas sur les slots libres de la page). Son principal avantage réside dans le contexte Client-Serveur : le serveur ne dépense pas de temps CPU à analyser le contenu des modifications validées afin de détecter une éventuelle corruption des méta-structures de la page.

Dans une première version, nous utilisons le page diffing, mais pour l'avenir, l'object diffing apparaît préférable.

---

21 Car comme cette opération intervient sur l'ensemble des pages modifiées en fin de transaction, ce débit s'améliore vers l'émergence de caches processeur de second niveau de grande capacité (actuellement 4Mo de RAM à 10ns pour MIPS R4000, DEC 21064 et T.I. SuperSparc).

22 Le coût CPU du Page Diffing varie linéairement en fonction du taux de modification de la page entre 220µs (resp. 120µs si caché) pour un taux de 0,1% à 315µs (resp. 210µs si caché) pour un taux de 100%. Il faut comparer ces valeurs avec le coût de 190µs (140µs si caché) pour le transfert d'une page de 4Ko dans un buffer. Ces mesures ont été effectuées pour une page de 4Ko avec un PPC601/60Mhz crédité d'une puissance crête de 56 SPECint92 dans les conditions d'un cache mémoire purgé (resp. dans les conditions où les images originales et modifiées des pages sont dans le cache mémoire).

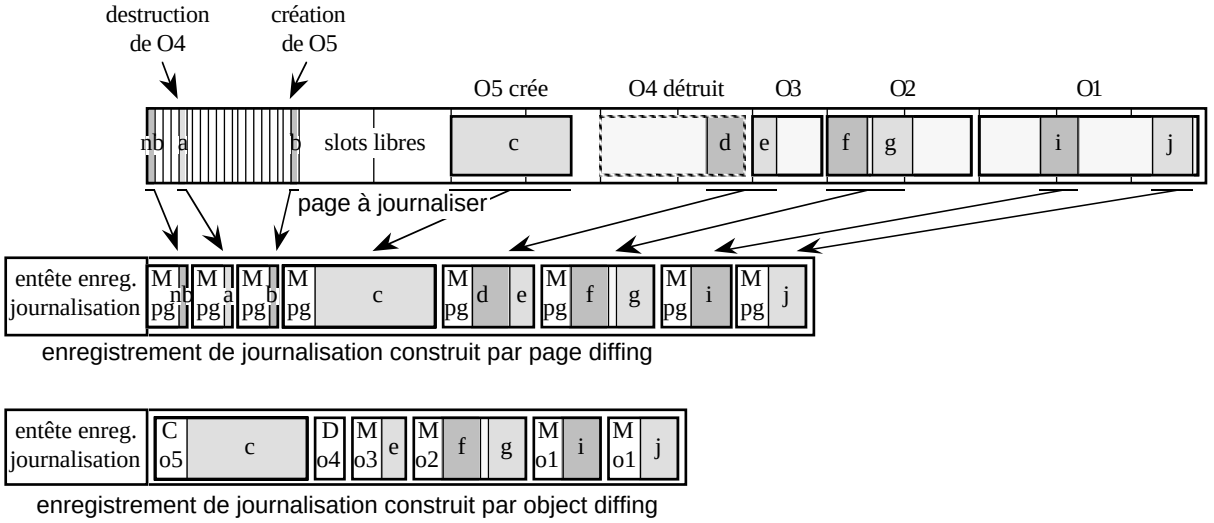
V.4.3. Opérations de Diffing et Journalisation

La structure de l'enregistrement de journalisation est commune aux deux techniques de diffing. L'enregistrement est préfixé par un entête qui contient les informations de séquences de la journalisation (i.e. numéro global dans l'organe de journalisation parallèle - section IV.3.2) et le numéro de la page journalisée. Le suffixe contient la liste des opérations sur les objets de la page. Pour l'object diffing, ces opérations sont les opérations déjà citées :

- création d'un objet accompagné de la nouvelle valeur,
- l'augmentation de la taille d'un objet accompagnée de la valeur de l'extension,
- la modification d'une partie de l'objet accompagnée de la valeur de la partie modifiée,
- la diminution de taille d'un objet,
- la destruction d'un objet avec ou sans réallocation du numéro d'objet.

Pour uniformiser la structure des enregistrements de journalisation entre les deux techniques, le page diffing ne considère que l'opération de modification sur un objet unique qui est la page. Les comparaisons dans les objets (ou la page) se font avec un granule minimum qui évite de subir le surcoût en place liée à l'entête de l'opération (8 octets). Dans l'exemple de la figure 5.18, l'objet O2 comporte 2 parties f et g modifiées séparées par une partie identique à l'ancienne. Les algorithmes de diffing incluent cette partie dans l'enregistrement de journalisation car elle est inférieure à une entête de champ d'opération. Par contre dans le cas de l'objet O1, les 2 modifications i et j sont trop "écartées" et les algorithmes de diffing génèrent 2 champs distincts. Remarquons que le page diffing enregistre les parties modifiées des objets détruits comme O4, et les changements dans la table des objets en entête de la page (nb,a,b) et que l'object diffing, réalisé objet par objet, ne tient pas compte des modifications mitoyennes.

Figure 5.18: Enregistrements de journalisation par page diffing et par object diffing

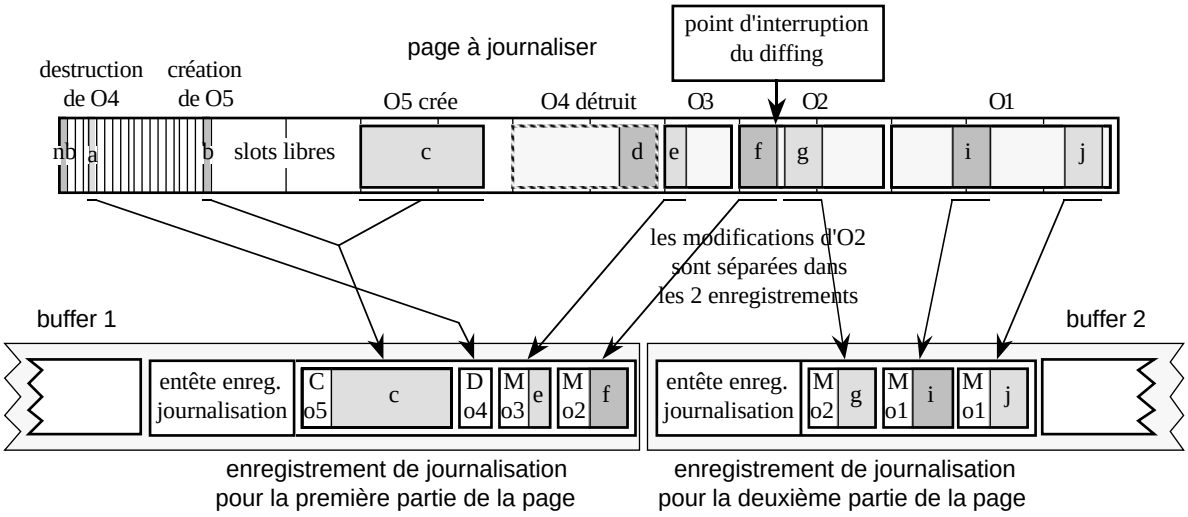


La journalisation d'une page ne génère normalement qu'un seul enregistrement contenant plusieurs champs, contrairement à la journalisation de QuickStore [White94] qui enregistre les différences de la page "diffée" dans des enregistrements séparés. Notre approche permet de :

- grouper les modifications opérées sur une page dans le même enregistrement ce qui en cas de journalisation parallèle facilite la reprise sur panne,
- et surtout d'économiser les entêtes d'enregistrement (≈ 50 octets pour QuickStore) en utilisant des entêtes de champ beaucoup plus petits (8 octets) à l'intérieur d'un entête d'enregistrement.

Toutefois, l'enregistrement de diffing d'une page peut être **fragmenté** pour améliorer le remplissage des buffers de journalisation (cf. section IV.3) ou de communication. La fragmentation intervient quand l'opération de diffing (Page ou Object), qui remplit un buffer, déborde de celui-ci. Le diffing clos alors l'enregistrement et le buffer rempli peut être journalisé ou émis. Le diffing de la page est continué dans un autre buffer (qui peut ne pas être vide) en créant un nouvel enregistrement contenant les différences trouvées dans le restant de la page. La figure 5.19 reprend la page modifiée de la figure 5.18 et lui applique l'object diffing : cette opération déborde au cours de la comparaison de l'objet O2, l'enregistrement est alors fermé. L'object diffing reprend ensuite la comparaison sur le restant de l'objet O2 et sur les restants en créant un deuxième enregistrement pour la page.

Figure 5.19 : Fragmentation de l'opération d'Object Diffing.



VI. L'interface Langage: le typage des objets.

L'interface Langage type les objets courts définis par la couche Objet. Le contenu de l'objet court est structuré en fonction des types qui sont défini avec l'interface Langage. Le langage que nous avons choisi pour cette interface est le C++, cependant cette couche reste indépendante des couches précédentes.

Nous commençons cette section en justifiant notre choix du langage C++ comme interface de WEA. Nous détaillons l'implantation de l'interface C++ et des outils associés. Elle offre l'implémentation standard du langage C++ utilisé pour la description du schéma et pour la manipulation de ces objets. L'interface Langage répertorie l'ensemble des classes C++ utilisées par les Espaces de Travail et instancie ces classes dans les objets courts définis par l'interface Objet.

VI.1. Choix du langage C++

Les Gérants d'Objets offrent des interfaces qui se situent à divers niveaux [Amiel92] :

- Les interfaces Programmables d'Application (API) sont des fonctions de bibliothèque qui permettent de manipuler les principales opérations du Gérant d'Objets. Ces fonctions sont appelées explicitement à l'intérieur d'un code écrit dans un langage quelconque. L'avantage de cette approche est la simplicité de réalisation du Gérant d'Objets et la possibilité d'utiliser plusieurs langages. Son inconvénient majeur est d'obliger le développeur à expliciter tous les mécanismes de gestion de la persistance utilisée : un exemple d'API est constitué par les premières versions d'ONTOS [Ontologic90].
- Les langages généraux de programmation offrent l'avantage d'uniformiser le code des applications non persistantes et celui des applications persistantes. Dans le domaine des bases de données orientées objet, plusieurs types de langage sont envisageables. Par exemple, Smalltalk est un langage interprété qui permet une définition extensive des propriétés des objets mais avec des performances variables. C++ offre une définition plus restrictive des objets [Joyner92] et des facilités de bas niveau (arithmétique de pointeurs, etc.), mais il offre aussi de très bonnes performances et un caractère quasi standard. Certains langages dérivés du C++ offre des propriétés intéressantes mais ils sont moins répandus. Enfin, les langages à base d'agents d'une part, à base de prototypes d'autre part, ne semblent pas être utilisées en Bases de Données.
- Les langages de requêtes offrent l'avantage d'une approche assertionnelle et d'une décomposition de requêtes optimisées. Mais, ils sont limités par l'absence d'une capacité de programmation générale [Kifer92].

Nous avons choisi d'offrir comme interface le langage C++ pour plusieurs raisons:

- Son caractère quasi standard dans l'industrie du génie logiciel met les développements non persistants ou persistants à l'abri de la disponibilité du langage sur une machine : le C++ a été choisi comme langage général par l'OMG.
- En tant que langage de programmation général, ses performances sont celles de son prédécesseur, le langage C. Une application développée sur un Gérant d'Objets doit pouvoir conserver les performances de son implantation non persistante.
- Le langage C++ offre la possibilité de redéfinir les opérateurs d'accès sur les instances de classes. Nous nous servons ces propriétés pour utiliser un compilateur du marché pour compiler les applications persistantes.

Néanmoins, la couche Langage C++ peut être interchangée avec une couche destinée à un autre langage. Nous envisageons de réaliser une couche Smalltalk structurant les objets suivant les principes de ce langage. Smalltalk possède plusieurs atouts dans le cadre de réseaux de machines hétérogènes ou mobiles. L'aspect interprété de ce langage permet à une application exécutée par un client d'importer les objets et les méthodes de leurs classes. Ce point est particulièrement important pour des clients PDA[23] (Personnal Digital Assistant) qui consultent des services sans disposer à l'avance de la couche présentation du service [Imielinski93b].

Nous verrons dans les sections suivantes que l'implantation de la couche Langage C++ se présente sous la forme d'un pré-compilateur très simplifié qui ne connaît pas le contenu des classes et d'un compilateur du marché qui structure le contenu des objets. Cette approche nous a permis de développer cette couche rapidement tout en utilisant les compilateurs se rapprochant le plus de la norme C++ [Ellis91b].

Cependant, cette implantation rend la structuration des objets fermement dépendante du langage utilisé et même du compilateur utilisé pour le même langage : les objets structurés par une couche Langage C++ ne sont pas exploitables par une autre couche Langage. Des propositions ont été faites pour décrire les structures des objets persistantes dans une formulation de la structure commune à l'ensemble des langages que proposent le Gérant d'Objets [Deux90][Carey94]. Mais cette approche nécessite d'écrire un compilateur, pour chaque langage, qui respecte la structure commune.

VI.2. Techniques d'implantation

La réalisation d'un compilateur C++ est un travail de très longue haleine pour offrir ce langage comme interface d'un Gérant d'Objets. La plupart des constructeurs commerciaux de Gérant d'Objets s'adressent à un constructeur de compilateur C++ pour réaliser le compilateur de leur langage d'interface : ObjectStore et ODE[24] utilise le compilateur C++ d'ATT et les produits universitaires comme Exodus et Texas ont adapté le compilateur GNU G++ de la Free Software Foundation. Cette approche rend l'interface du Gérant d'Objet dépendante du type du compilateur ou de ces versions. Ces compilateurs ne suivent pas toujours l'intégralité des règles du C++ [Ellis91b]. D'autre part, les outils, qui sont utilisées lors du développement d'applications non persistantes comme les debuggers ou les profilers, ne sont pas toujours réutilisables avec le compilateur adapté pour les besoins du Gérant d'Objets.

Nous proposons une approche portable qui rend l'interface Langage C++ indépendante de l'origine ou de la version du compilateur auquel elle est associée; les outils attachés au compilateur restent utilisables comme lors du développement de l'application non persistante. Cette approche consiste à utiliser les fonctionnalités du langage C++ : le langage C++ permet de redéfinir ("surcharger") certains opérateurs comme les opérations d'accès aux objets ou encore les opérateurs d'allocation des objets. Nous utilisons ces fonctionnalités afin de définir pour chaque classe C++ de l'application les méthodes ad hoc qui permet :

---

23 Les concepteurs du Newton d'Apple ont choisi un langage de frame comme plate-forme de développement des applications de ce PDA [Smith94]. Cette approche rend l'application qui est interprétée du processeur du client.

24 ODE est réalisé par une équipe d'ATT.

- l'allocation d'instances persistantes de classes C++,
- l'appel des méthodes virtuelles associées,
- la déréférenciation des références persistantes vers les objets.

Ces fonctions sont générées par un pré-compilateur qui analyse uniquement le schéma de la base. Ce schéma se compose des définitions des classes C++ définies dans l'application.

VI.2.1. Allocation des instances persistantes des classes C++.

Les instances des classes sont allouées dans les objets courts de la couche Objet. L'allocation est réalisée par la surcharge de la méthode `new`(). Ordinairement, cette méthode permet l'allocation dynamique d'objets dans le tas (heap) de l'application non persistante; cette méthode prend la taille des objets de la classe à instancier comme paramètre et retourne l'adresse d'une zone mémoire de la taille demandée. L'appel à la méthode new() est suivi de l'appel d'une des méthodes "constructeur" de la classe qui initialise le contenu de la zone mémoire allouée.

L'allocation consiste à redéfinir cette méthode new() pour chaque classe du schéma de la base. La méthode new() redéfinie consiste à allouer un objet court dans une page de la base, de mapper le segment qui contient cette page et de retourner l'adresse de début de la partie utile[25] de l'objet court alloué. Cette méthode ajoute également, dans l'entête de l'objet, l'identifiant de la classe à laquelle appartient l'instance qui va être créée dans l'objet court. Cette identifiant est un numéro unique sur l'ensemble des classes analysées par le pré-compilateur; nous verrons dans la suite que cet identifiant sert à appeler les fonctions appelées lors de l'installation d'une page d'objets courts dans l'espace d'adressage ou lors de la validation des pages modifiées.

Les surcharges de l'opérateur new() pour une classe de schéma `my_class` accessible au développeur sont les suivantes :

- `my_class* new(size_t, WEA_TALLOC_t);`

  cette surcharge permet de distinguer si l'objet court, contenant l'instance de la classe `my_class`, doit être créé persistant ou temporaire (cf. section V.2) suivant la valeur du paramètre `WEA_TALLOC_t`.

- `my_class* new(size_t);`

  cette surcharge ne permet que l'allocation persistante de l'objet court.

- `my_class* new(size_t, p_Object nearby);`

  cette surcharge permet que l'objet court soit créé à proximité (cf. section V.2) de l'objet court pointé par la référence persistante `nearby`.

---

25 partie consécutive de l'entête de l'objet court.

Notons enfin que la taille d'une instance de classes C++ ne peut pas dépasser celle d'une page contenant des objets courts. Cette contrainte est contournée par le devellopeur en définissant la classe de façon hiérarchique et en utilisant les types complexes fournis par WEA. Dans l'exemple suivant, le membre `tab` de classe `Huge` est un tableau de caractère de grande taille; la contrainte de taille est contournée par l'utilisation de la classe String appartenant à la bibliothèque de classes fournies par WEA.

```
class Huge { // originale
 public:
    int   size;
    char  tab[100000];
}; // sizeof(Huge) > 4096
```

```
class Huge { // redéfinie
 public:
    int   size;
    String tab; // structure dynamique
}; // sizeof(Huge) < 4096
```

VI.2.2. Accès aux méthodes virtuelles

Dans le langage C++, une méthode est une fonction membre d'une classe qui ne peut être appelée que pour une instance de la classe. Une méthode d'une classe est héritée par les classes dérivées qui peuvent redéfinir ces méthodes; cette redéfinition correspond en général à la spécialisation de la méthode. Les langages orientés objet permettent de manipuler les instances de classes sans se soucier des spécificités de celles-ci. En C++, cette manipulation se fait par les méthodes virtuelles.

Le C++ autorise deux types de méthodes : les méthodes virtuelles et les méthodes non virtuelles. La virtualité fournit le mécanisme d'abstraction décrit précédemment lors des appels aux méthodes. Les secondes correspondent à une optimisation des premières quand la virtualité n'est pas nécessaire. Les compilateurs C++ implantent la virtualité en ajoutant dans la structure de chaque instance un pointeur vers la table des méthodes de sa classe. La figure 5.20 représente les structures des deux instances des classes `Base` et `Derive`. Le pointeur `_vtbl`, ajouté dans la structure de l'objet, est inaccessible au développeur.

La table de méthodes virtuelles d'une classe est une donnée statique des applications. Son adresse varie d'un programme à l'autre. Dans le cadre des instances persistantes, ce pointeur est stocké dans les objets courts. Il convient d'initialiser ce pointeur dans l'objet court par rapport au programme courant qui accède à la page persistante contenant l'objet. Le pointeur `_vtbl` est invisible dans le langage C++, cependant il est possible de réinitialiser ce pointeur vers la table des méthodes du programme courant en utilisant une propriété du constructeur de classe vide : le constructeur vide n'effectue que la réinitialisation du pointeur caché[26] `_vtbl` pour une zone mémoire dans la pile quand l'instance est une variable ou un paramètre de fonction, ou bien dans le tas à la suite d'un appel à l'opérateur `new`(). Le pré-compilateur génère donc une fonction de réinitialisation[27] du pointeur vers la table des méthodes virtuelles pour toutes les classes du schéma; ces fonctions sont appelées sur tous les objets courts d'une page quand celle-ci est accédée pour la première fois dans l'Espace de Travail. L'identifiant de classe,

---

26 En fait, la structure d'une instance comporte plusieurs pointeurs _vtbl en cas d'héritage multiple ou en cas d'agrégation de classes. Le constructeur vide réinitialise chaque pointeur dans les deux cas. Dans le deuxième cas, le constructeur vide de l'instance "englobante" appelle récursivement ceux des instances agrégées.

27 Cette fonction est compilée séparément pour éviter que le constructeur vide de réinitialisation n'interfère avec le constructeur vide défini par le développeur. Cette technique est aussi utilisée pour générer les fonctions d'unswizzling et de propagation de la fonction setPersistent() dans les objets composites.

contenu dans l'entête de l'objet court, sert à appeler la fonction correcte pour l'instance contenue.

Cette méthode permet d'ignorer la structure d'une instance et la localisation de la table des méthodes virtuelles de celle-ci dans l'espace d'adressage; on peut ainsi garantir la portabilité de l'interface quel que soit le compilateur utilisé pour compiler le code de l'application. Cependant, cette méthode a un inconvénient pour le mapping des pages accédées. L'opération de réinitialisation modifie le contenu physique d'une page; la page mappée est réécrite sur disque quand elle est réquisitionnée par le noyau, bien que les objets de cette page aient été seulement accédés en consultation par la transaction qui la mappe. Cette réécriture intervient seulement pour la première réquisition de la page. La solution consiste à utiliser les identifiants de classes pour accéder aux tables des méthodes virtuelles. Mais cette solution implique de pourvoir redéfinir l'appel aux méthodes virtuelles dans le compilateur C++.

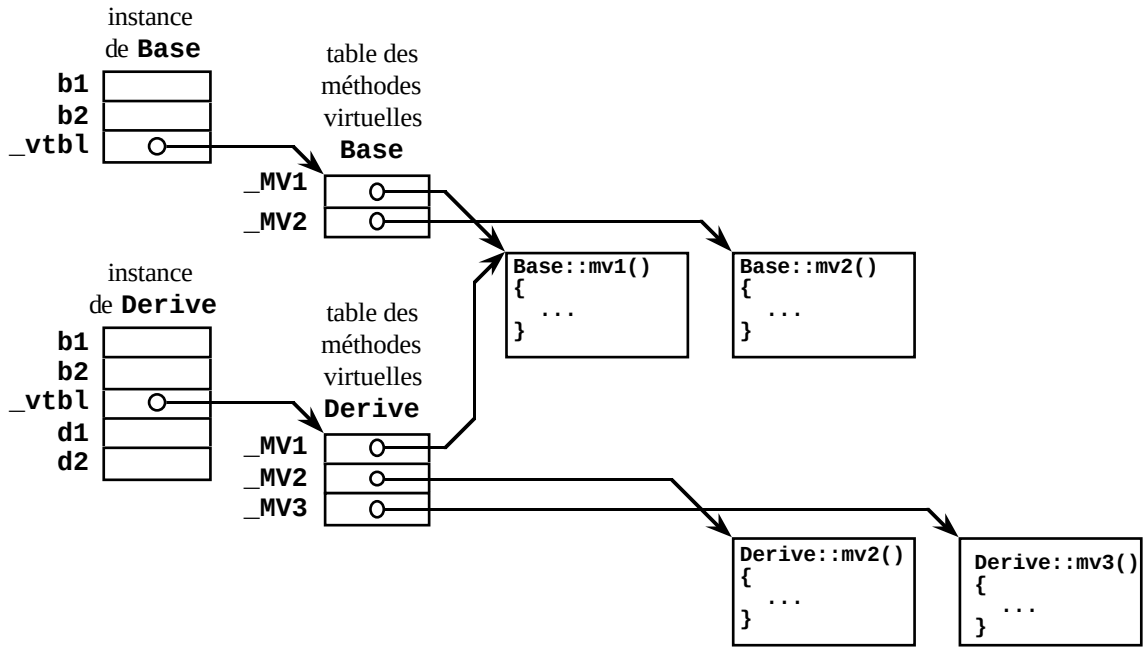
Figure 5.20 : Instances de classes et Tables des Méthodes Virtuelles.

```
class Base { public:
     int b1,b2;
     ms();
 virtual mv1();
 virtual mv2();
};
class Derive { public:
     int d1,d2;
     ms();
 virtual mv2();
 virtual mv3();
};
```

```
Derive d;

Base*  pb=&d; // pb et pd referencent
Derive* pd=&d; // une instance de Derive

pb->ms(); // Base::ms()
pd->ms(); // Derive::ms()
pb->m1(); // Base::m1()
pd->m1(); // Base::m1()
pb->m2(); // Derive::m2()
pd->m2(); // Derive::m2()
```



VI.2.3. Références Persistantes et Déréférenciation

Les instances de classes C++ peuvent contenir des références (pointeur) vers d'autres instances. Dans le contexte d'instances persistantes, les références entre instances doivent être également persistantes; les références persistances sont donc naturellement implantées par les identifiants d'objets.

Les références persistantes sont des instances de classes générées par le pré-compilateur. Ces classes sont définies pour remplacer l'usage des types pointeurs dans les définitions des classes, des méthodes et des fonctions de l'application persistante. L'identificateur d'une classe référence est l'identificateur de la classe persistante préfixé par `p_`. Le type pointeur `myclass*` vers les instances de `myclass` devient la classe référence `p_myclass`. Les opérateurs d'accès de classes références sont redéfinies par le pré-compilateur pour appeler l'opération de déréférenciation.

L'opération de déréférenciation de la couche Langage n'ajoute que le déplacement correspondant à l'entête dans l'opération de traduction d'objets de la couche Objet (cf. section V.3.2). Cette opération est lente par rapport à un accès par pointeur non persistant. Nous proposons deux techniques pour accélérer cette opération :

- Cache de références persistantes.

  Chaque thread possède un cache privé des "dernières" déréférenciations effectuées par la thread. Ce cache est une table avec un nombre d'entrées fixes gérée à la manière des caches "Direct-Mapped" des processeurs DEC Alpha et MIPS R4000 : l'adresse associée à un identifiant ne peut être rangée que dans l'entrée du cache désignée par le résultat du hachage de l'identifiant. La recherche dans le cache est une fonction relativement rapide[28]. Cependant dans le contexte multi-threads, le temps nécessaire pour obtenir l'adresse de la table privée[29] ralentit considérablement le processus de cache.

- Swizzling de références persistantes.

  Le *swizzling at dereferenciation* [White92] consiste à remplacer, dans la référence persistante, l'identifiant par l'adresse rendue par la première déréférenciation; les déréférenciations suivantes utilisées directement avec l'adresse de l'objet. L'opération inverse (*unswizzling*) doit être effectuée au moment de la validation des pages modifiées sur l'archive pour convertir les adresses en identifiant d'objets.

  Contrairement aux techniques de swizzling utilisé dans ObjectStore et Texas, cette technique ne *swizzle* que les références parcourues et n'*unswizzle* que les références parcourues dans les objets des pages modifiées. Par contre, elle ajoute un léger

---

28 La recherche dans le cache ajoute un surcoût de 12 instructions à un accès classique par pointeur sur un Sparc.

29 Initialement, les concepteurs de la bibliothèque de threads de Solaris avaient prévu de réserver un registre du processeur pour pointer la zone de données privées de la thread courante ; une seule instruction eut été nécessaire pour accéder à une entrée du cache. Malheureusement pour suivre les dratfs POSIX, les concepteurs ont été amenés à fournir l'accès aux données privées sous la forme d'une fonction dont le coût est approximativement de 140 instructions !

surcoût qui correspond au test[30] de la référence (i.e. est-elle swizzlée ou non ?) comme le montre le code de l'opérateur d'accès surchargé :

Cette technique possède un autre avantage dans le contexte du mapping : la première déréférenciation appelle explicitement le Gérant d'Objet pour obtenir le verrou pour la consultation de la page (verrou partagé); les mappings privés peuvent être ainsi initialement protégés contre la modification. La première consultation n'entraîne pas ainsi de violation des protections dont le coût de traitement doit être ajouté à celui de la modification des protections sur la page[31].

Une fonction d'*unswizzling* des références persistantes est générée par le pré-compilateur pour chaque classe du schéma. Le pré-compilateur utilise les propriétés récursives des méthodes "constructeur" : le constructeur de l'instance appelle le constructeur de chaque référence persistante qui appelle la méthode d'unswizzling de celle-ci.

```
inline MyClass* p_MyClass::operator->() {
 if( this->_union.flag == TYPEOID ) // test ajouté
   // cas où la référence n'est pas swizzlé e : appel à la couche Objet
   return (MyClass*) getObjet(this->_union.OID);
 else
   // cas où la référence est déjà swizzlée (survoût de 4 instructions)
   return (MyClass*) this->_union.addr;
}
```

La technique du swizzling n'est utilisée que dans le contexte de clients mono-transaction. En effet dans le contexte d'un Espace de Travail multi-transaction, le swizzling du pointeur est seulement valide pour une thread. Il doit donc être réalisé dans le mapping privé de chaque thread : ceci a comme effet de déclencher le mécanisme de copy-on-write et donc les threads ne se partagent plus les pages physiques contenant des données consultées. Dans le contexte du client mono-transaction, le swizzling peut être réalisé sur les mappings partagés des segments image. Le swizzling est conservé d'une transaction à l'autre pour les pages consultées tant que la page n'est pas invalidée du cache client; les pages modifiées sont unswizzlées par le client avant d'être redescendues au serveur.

L'unswizzling nécessite une table de correspondance assurant la traduction de l'adresse d'un objet vers son identifiant; cette table est construite au fur et à mesure des déréférenciations. Notre implantation courante est simplifiée par le fait que l'entête des objets courts contient l'identifiant de l'objet : nous n'utilisons donc pas de table (l'unswizzling va directement chercher l'identifiant d'un objet référencé dans son entête). Cependant, cette simplification provoque des mouvements de pages entre la mémoire et le disque si les pages contenant les objets accédés ont été évincées de la mémoire.

La technique de cache de déréférenciation est donc utilisé dans les autres cas et en particulier par les services d'Opération et les services Mixtes. Dans ce dernier cas, le client peut utiliser la

---

30 Avec un compilateur pour Sparc, ce test ajoute 4 instructions supplémentaires une fois que la référence est swizzlée (un codage manuel ramène le test à 3 instructions).

31 La manipulation du signal prend 380µs; le changement de protection 130µs pour une page de 4Ko (et 260µs pour un segment de 1024 pages) sur une Sun IPC (13,8 Spec Int92). L'utilisation de la violation pour détecter les verrous en lecture ajoute donc un surcoût de 7000 instructions par page consultée; soit l'équivalent de 1750 test dans le swizzling "at dereferenciation".

technique du swizzling pendant que le serveur utilise la technique de cache; on doit gérer deux versions de code compilé pour chaque méthode, une pour la thread de service et une pour l'application client.

VI.2.4. Déréférenciation et Verrouillage

Du point de vue de l'interface, le verrouillage est transparent au développeur comme le montre la section de code suivante (figure 5.21). Cependant celui-ci dispose de la méthode `p_<class>::update()`, définie pour toute classe référence, qui permet de demander explicitement un verrou en écriture; on évite ainsi le surcoût de la violation des protections.

Pour les pages d'objets courts, le verrouillage en lecture est demandé par l'opération de déréférenciation (par cache ou par swizzling) quand l'identifiant n'est pas dans le cache de déréférenciation ou si la référence n'a pas été swizzlée. Le verrouillage en écriture est par contre demandé par le mécanisme de gestion des violations lors la thread viole les protections mémoires. Dans le cas des objets longs, le verrouillage (lecture et écriture) est appélé page par page quand la transaction accède à ces pages. Le verrouillage est appelé par le mécanisme de gestion des violations

Figure 5.21 : Verrouillage transparent des pages d'objets.

```
p_Person adam = ...  // une référence persistante (non déréférencée)
            // est affectée à adam

cout << adam->nom;   // verrouillage en lecture de la page (A)
            // contenant l'objet référencé par adam

p_Person eve = adam->epouse; // affectation de référence :
               // pas de verrouillage

cout << eve->nom;   // verrouillage en lecture de la page (E)
           // contenant l'objet référencé par eve

eve->age++;      // violation des protections mémoires :
           // verrouillage en écriture de la page (E)

adam.update()->age++; // permet d'éviter une violation
           // des protection sur la page (A)
```

VI.3. Bibliothèque de Classes Persistantes.

La couche Langage C++ fournit une bibliothèque de classes permettant de créer des collections d'instances (listes, vecteurs, ensembles) ou de longues chaînes de caractères. Cette bibliothèque contient également les fonctions de manipulation de la métabase. Toutes ces classes sont des classes persistantes générées par le compilateur.

VI.3.1. Métabase.

La base ne contient qu'une seule instance créée au moment d'initialisation de la base. Cette instance possède le premier identifiant d'objet court alloué dans la base. Actuellement, cette

instance ne contient que la racine de persistance. Les informations concernant le schéma seront ajoutées ultérieurement.

La racine de persistance constitue le point d'entrée d'une application dans la base : elle permet de désigner des instances particulières avec un nom externe (une chaîne de caractères). La racine fait la correspondance entre ce nom et la référence persistante d'une instance; normalement depuis cette instance, l'application peut atteindre un sous-graphe du graphe d'instances qui constituent la base.

L'application crée un nouveau point d'entrée au moyen de la méthode `setName`() définie pour chaque objet; une autre application accède à l'instance ainsi référencée au moyen du constructeur des références persistantes. Ce constructeur contrôle[32] si la classe de l'instance référencée dérive de la classe des instances qu'il référence. Dans l'exemple 5.22, la transaction 1 crée une nouvelle instance de la classe `Chien` et attache un nom externe à cette instance. La transaction 2 accède à l'instance référencée par ce nom externe : la référence persistante `pl` n'est pas initialisée car l'instance référencée n'hérite pas de la classe `Loup`.

Figure 5.22 : Racines de Persistance

```
{ // corps de la transaction 1
 p_Chien pc = new Chien;
  ...
 pc->setName("AmiDeLHomme");
}
```

```
{ // corps de la transaction 2
 p_Animal  pa("AmiDeLHomme"); // OK
 p_Mammifere pm("AmiDeLHomme"); // OK
 p_Loup   pl("AmiDeLHomme"); // KO
  ... // suite
}
```

VI.3.2. Classes Composites.

L'instance d'une classe composite est constituée de plusieurs objets courts qui stockent les fragments de la structure de l'instance. L'instance composite se compose d'un objet racine qui donne son identifiant à l'instance composite et d'objets noeuds dont les identifiants peuvent être réalloués quand ils sont détruits. Les méthodes associées à cet objet composite doivent être déclarées et définies dans la classe de l'objet racine.

Du point de vue du pré-compilateur, les classes composites n'existent pas : celles-ci sont implantées au moyen des classes persistantes qui instancient le contenu de l'objet racine et celui des objets noeuds. Le pré-compilateur génère pour ces classes les fonctions vues précédemment. Néanmoins comme le pré-compilateur ne prend pas en compte le contenu sémantique de l'objet, ces fonctions peuvent ne pas être optimisées pour le contenu de l'objet. Prenons l'exemple d'un noeud d'index dans une instance composite de BTree; ce noeud est un tableau de références persistances vers des noeuds inférieurs : la fonction d'*unswizzling* parcourt l'ensemble des références alors que seules quelques références de têtes pointent réellement sur des objets. Le développeur peut redéfinir les différentes fonctions générées par défaut par le pré-compilateur afin d'optimiser celles-ci.

La racine d'une instance composite peut être créée dans une de ces trois zones :

- la zone "persistante" (i.e. dans la base);
- la zone "temporaire" de la thread (i.e. le tas privé de celle-ci)

32 Le pré-compilateur génère des fonctions de test de l'héritage sur les identifiants de classe.

- sur la pile de la thread (i.e. l'instance est une variable ou un paramètre d'une méthode ou d'une fonction).

Le concepteur d'une classe composite doit tenir compte de la zone d'instanciation de la racine pour définir les constructeurs des différentes classes composantes. La couche Page fournit une fonction permettant de tester si un objet se trouve dans une des trois zones.

Les classes composites définies dans la bibliothèque sont :

- les Collections Génériques d'instances persistantes,

  Elles permettent des structures des instances des classes persistances en listes, vecteurs, ou index. Leur définition dans le schéma fait l'objet du mot-clé réservé `colldef` qui rend leur manipulation générique.

- la classe String,

  La classe String instances de chaînes de caractères de taille moyenne. Elle est implantée comme un arbre équilibré dont les feuilles contiennent les fragments de la chaîne. Si la chaîne est de petite taille, elle est stockée dans la racine.

- la classe Large.

  Cette classe correspond à de Grands Objets (Larges Objects ou LAOB) définie dans Ode, Exodus et Starburst [Biliris92, Biliris93] destiné à recevoir de longues chaînes de caractères (supérieures à la taille d'une page). Ces objets sont composés d'objets longs, et la contiguïté est assurée normalement entre les pages à l'intérieur d'un objet long (cf. section V.1.2). Les objets courts implantent la racine et les noeuds d'index vers les objets longs. Cette structure offre un compromis entre le BLOB difficile extensible et l'instance de String qui nécessite de nombreux niveaux d'index. La contiguïté dans les blocs de pages permet d'améliorer le temps d'accès aux disques en favorisant des lectures séquentielles [Reddy94].

VI.4. Outils de l'interface C++.

La chaîne des outils, qui constitue l'interface C++, est (figure 5.23):

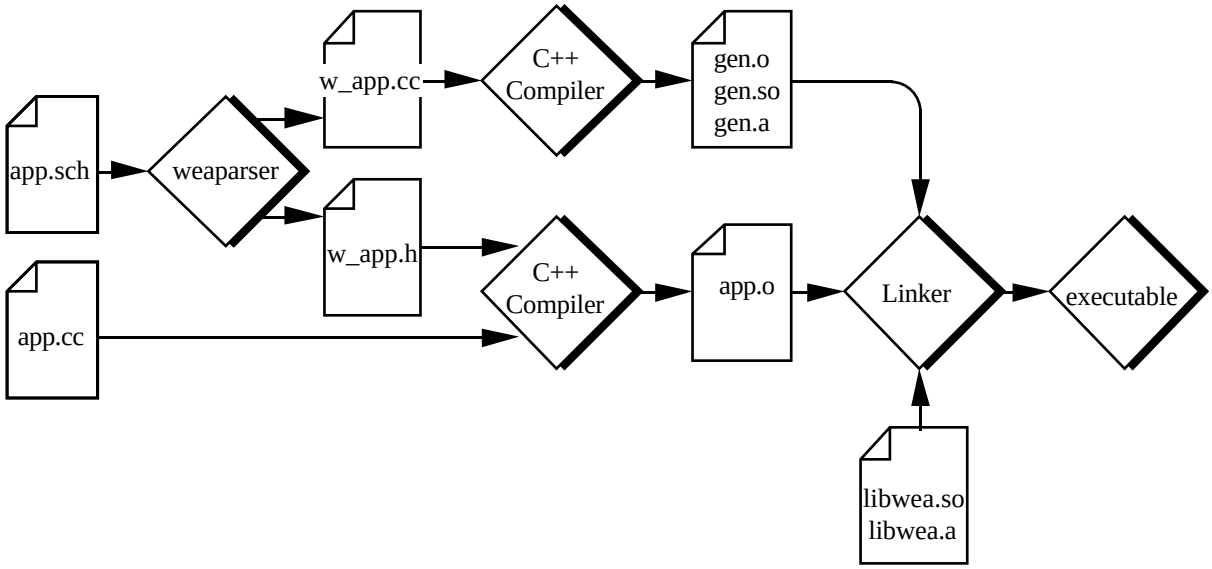
- le pré-compilateur de schéma,
- un compilateur C++ du marché et l'éditeur de lien du système hôte.

Le pré-compilateur analyse les classes du schéma de la base (fichier app.sch). Il génère alors, pour chaque classe, la classe référence persistance associée (fichier w_app.h) et ajoute les méthodes nécessaires à la persistance à la classe analysée (fichiers w_app.h et w_app.cc). Chaque classe référence, qui redéfinit les opérateurs d'accès, est compilée avec l'application (fichiers app.cc). Les méthodes associées à la persistance (création, destruction, swizzling des _vtbl, unswizzling des références persistances) sont compilées dans des fichiers séparés de l'application (car les définitions multiples des constructeurs vides utilisés pour ces méthodes sont conflictuelles lors de la compilation). Ses méthodes sont ajoutées dans l'application à l'édition de lien.

Le compilateur C++ est un compilateur quelconque du marché (ATT ou GNU). Il génère le binaire des applications à partir du code défini par le développeur et à partir des méthodes générées par le pré-compilateur. La pré-compilation permet d'utiliser des compilateurs existants et de profiter des outils de développement qui leur sont généralement associés (debugger, browser de classe, analyseur de performance, ...). Le second avantage découle du premier en évitant la définition d'un langage semblable C++ mais s'écartant quelque peu de la norme. On évite ainsi de définir une nouvelle interface propriétaire.

La chaîne de compilation se termine par l'édition de lien de l'application avec les fonctions des couches précédentes (couche Page et couche Objet). L'application devient un Espace de Travail capable se s'abonner aux services du chapitre 4 et/ou de publier des services.

Figure 5.23 : Chaîne de Compilation de l'interface Langage C++.



VII. Versions Alternatives et Travail Coopératif

Nous avons proposé dans la section IX du chapitre 4 notre principe de fonctionnement du travail coopératif sur une base de données : le travail coopératif débute une phase de production, de consultation et de modification de versions de la base par des transactions coopératives, et se termine par une phase d'élaboration de la nouvelle base par l'activité superviseur. L'implantation du travail coopératif repose essentiellement sur la gestion des versions alternatives et sur les mécanismes d'accès aux objets dans ces versions par les transactions coopératives.

Nous rappellerons les règles de manipulations des versions alternatives. Ces règles permettent d'implanter les versions avec la technique de "delta". Ensuite, l'opération de déréférenciation ajoute un niveau de décodage supplémentaire mais la consultation des objets d'une version utilise les techniques de mapping présentées dans les sections précédentes. Nous verrons ensuite la construction des versions alternatives au moment des validations des transactions coopératives. Nous terminerons sur l'élaboration de la version définitive qui

consiste à modifier les objets de la base en fonction de ceux qui ont été modifiées dans les versions.

VII.1. Transactions Coopératives et Versions alternatives

VII.1.1. Règles de manipulation des versions alternatives

La transaction coopérative spécifie la version alternative de la base sur laquelle elle souhaite travailler avant de commencer des consultations ou des modifications dans la version alternative. Les modifications ne sont possibles que si la version est modifiable. Dans le cas où la version est stable, les modifications ne sont plus possibles sur la version; en revanche, une transaction peut dériver une nouvelle version modifiable à partir de cette version stable et apporter ensuite des modifications à la nouvelle version. L'ensemble des dérivations définit un arbre de versions dans lequel la racine est la base de données, considérée comme la première version stable; les noeuds internes sont des versions stables tandis que les feuilles sont des versions stables ou modifiables.

VII.1.2. Implantation de la version par une technique de "delta"

Ces règles de dérivation des versions nous permettent d'implanter les versions alternatives avec la méthode de "deltas" [Cellary90] : cette méthode consiste à ne stocker dans la version que les différences avec la version dont elle est dérivée. Cette implantation est rendue possible car une version n'est dérivable que si elle est stable. La version alternative est implantée avec un objet composite de la couche Langage : cet objet composite "delta" est constitué d'un objet "table de correspondance" et d'objets "valeur" contenant la nouvelle valeur d'objets modifiés dans les versions précédentes ou dans la base. La table de correspondance fait la correspondance de l'identifiant d'un objet modifié vers un objet "valeur" qui contient la nouvelle valeur.

La table de correspondance correspond aussi à la liste des objets modifiés par rapport à la base. L'objet composite est composé également d'une liste d'identifiants d'objets créés dans la version et une liste des identifiants des objets modifiés dans la version par rapport à la version précédente. Ces listes sont utilisées lors des consultations par les transactions coopératives, dans le processus de modération et celui d'élaboration par l'activité superviseur.

Nous distinguerons les deux stratégies de construction des objets composites. Les sous-objets "table", "liste" et "valeur" qui composent la version alternative, peuvent :

- soit être" regroupés" dans des pages afin que les versions ne se partagent pas de pages,

  Ce regroupement facilite essentiellement l'opération de diffing utilisé pour détecter les objets modifiés dans une version (cf. section VII.3), et la création de nouveaux objets. En revanche, il impose la taille minimale d'une page pour stocker une version alternative.

- soit être "mélangés" dans des pages communes.

  Le mélange des sous-objets permet de réduire la taille des versions; par contre, la création de nouveaux objets dans une version doit instancier les objets dans une zone temporaire de la transaction pour éviter le blocage des autres transactions qui consultent d'autres versions (cf. section VII.3).

Toutefois, ces deux stratégies séparent les sous-objets des "deltas" et les objets de la base en deux groupes de pages. Cette séparation facilite la désallocation des "deltas" à la fin du travail coopératif.

VII.2. Déréférenciation et Consultation des objets.

La spécification de la version alternative de travail par la transaction coopérative est en fait une demande envoyée à l'activité superviseur. Cette dernière accompagne sa réponse de l'identifiant de l'objet composite qui représente la version[33]. La thread, qui exécute la transaction coopérative, utilise cet identifiant comme point d'entrée sur la version alternative; elle mappe en privé le segment qui contient l'objet composite. Le travail peut alors commencer à partir des racines de persistances.

La déréférenciation par les transactions coopératives est réalisée en deux phases :

1- La première phase utilise la table de correspondance contenue dans l'objet composite. S'il existe une entrée pour l'identifiant d'objet à déréférencer, l'entrée contient soit l'indication que l'objet a été détruit, soit l'identifiant d'un objet "valeur"[34] qui contient une valeur modifiée de l'objet par rapport à celle de la base. L'identifiant, contenu dans l'entrée, est transmis à la phase 2.
   Si aucune entrée ne concerne l'identifiant à déréférencer, c'est que l'objet n'a jamais été modifié dans cette version ou dans une version précédente. L'identifiant de l'objet est donc transmis tel quel à la phase 2. Ce niveau correspond à un niveau logique d'identification.

2- La deuxième phase correspond au mécanisme classique de déréférenciation qui traduit, l'identifiant fourni par la première phase, vers l'adresse de l'objet dans un des mappings privés de la thread.

Comme les objets qui composent la version, sont des objets définis dans la couche Langage, la technique de swizzling peut être utilisée pour accélérer les déréférenciations si l'Espace de Travail le permet (une seule transaction coopérative dans l'Espace de Travail). Les références à swizzler sont celles contenues par le sous-objet "table" (y compris les identifiants contenues dans les entrées de la table), par les sous-objets "valeur" et par les objets de la base.

Une fois que l'identifiant est déréférencé, la transaction coopérative accède aux sous-objets de l'objet composite et aux objets de la base au travers des mappings privés des segments contenant les objets.

---

33 Cet identifiant est celui de la racine de l'objet composite qui est l'objet "table de correspondance".

34 L'objet "valeur" est typé par la classe de l'objet dont il représente la nouvelle valeur : il n'existe pas de type "valeur" enregistré par l'interface Langage.

La version alternative V2 de la figure 5.24 nous donne un exemple du contenu de la table de correspondance. L'entrée correspondant à l'identifiant de l'objet **A** contient une référence sur un objet identifié par **A"** et contenant la nouvelle valeur **A••** qui a été modifiée dans cette version. L'entrée correspondant à l'identifiant de l'objet **B** contient une référence sur un objet **B'** appartenant à une version précédente. L'entrée correspondante à l'objet **C** contient l'indication que cet objet a été détruit dans cette version ou une version précédente. Nous verrons dans la section suivante que les objets créés ont des identifiants particuliers. Dans l'exemple de la version V1, l'objet créé, qui contient la valeur **b•**, est référencé des autres objets par l'identifiant particulier β' (cf. figure 5.24).

VII.3. Modification d'une version alternative

Une transaction coopérative peut consulter plusieurs versions alternatives durant son exécution; à chaque fois, elle spécifie la nouvelle version qu'elle souhaite consulter. La transaction peut également modifier la version (si l'activité superviseur l'autorise). Une fois qu'une version spécifiée subit des modifications par la transaction, celle-ci ne peut plus changer de version jusqu'à la validation.

Les modifications d'une version sont réalisées dans les mappings partagés de segments qui contiennent les sous-objets du delta et les objets de la base. La validation de ces modifications par la transaction distingue deux cas :

- la version modifiable n'a jamais été modifiée depuis sa dérivation,
- la version modifiable a déjà été modifiée.

VII.3.1. Version nouvellement dérivée

Quand une version stable est dérivée en une nouvelle version modifiable, la dérivation ne crée pas l'objet composite correspondant à la nouvelle version. Celui-ci ne sera créé qu'au moment de la validation de la première transaction qui modifie la version.

La spécification de consultation ou de modification de cette version nouvellement dérivée retourne en fait l'identifiant de l'objet composite représentant la version stable dont dérive la nouvelle version. La transaction coopérative peut commencer ses consultations et ses modifications dans les mappings privés des segments contenant les objets de la version précédente; la version stable est préservée des modifications par les mappings en mode privé.

La validation de la transaction coopérative construit un nouvel objet composite contenant une nouvelle table de correspondance et de nouveaux objets "valeur" contenant les valeurs de tous les objets qui ont été modifiés dans les mappings privés. Cette allocation retardée de l'objet composite permet de construire celui-ci en connaissant ce qu'il va contenir; les structures des objets "tables" et "listes" peuvent être ainsi optimisées en fonction du contenu de la version, et les objets peuvent être regroupés dans les mêmes pages.

L'opération d'Object Diffing (cf. section V.4.2) est utilisée pour détecter les objets qui ont été modifiés ou détruits dans les mappings privés; par contre, les pages modifiées dans les mappings privées ne contiennent pas d'objets créés lors de la transaction.

Figure 5.24 : Structure des Versions alternatives

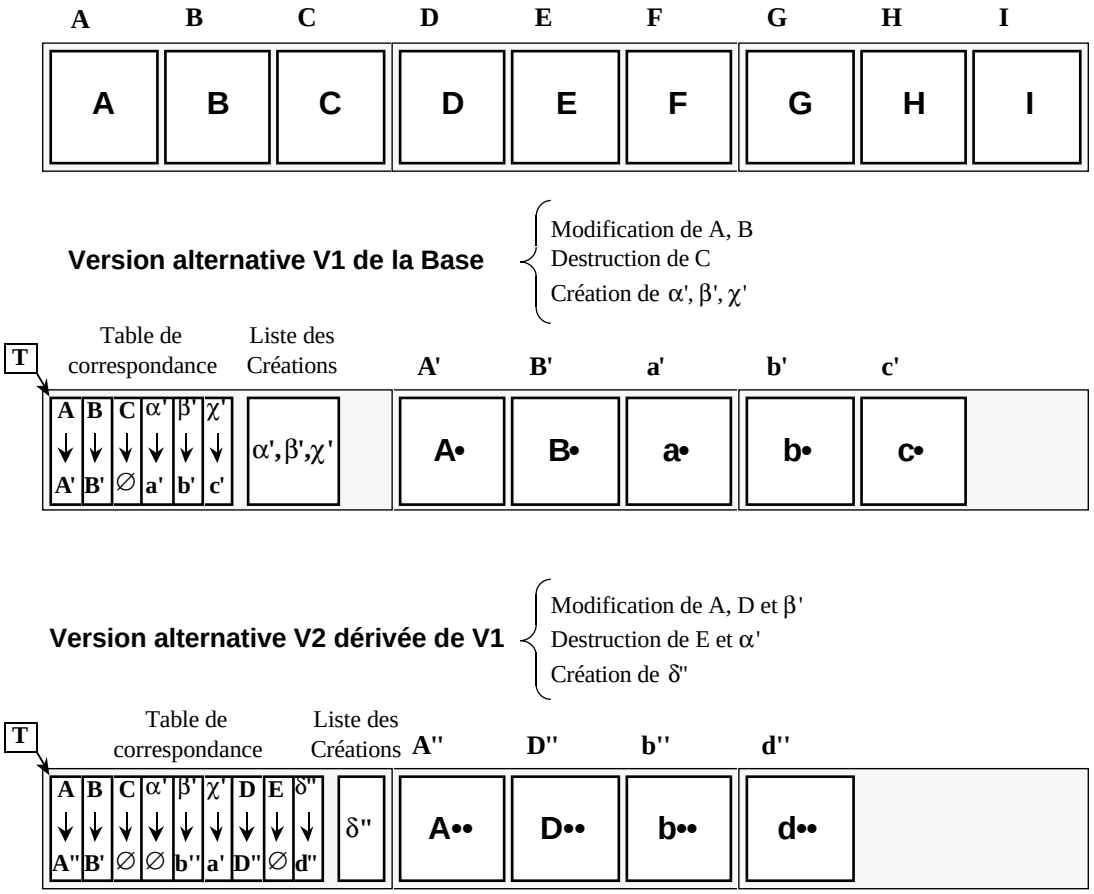
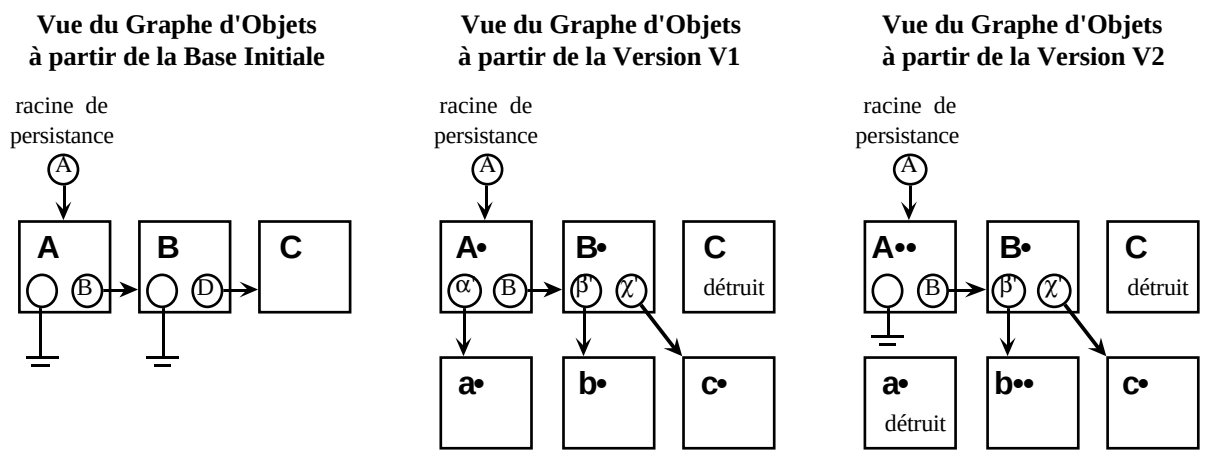


Figure 5.25: Graphe d'objets vu des différentes versions alternatives



La création d'objets nouveaux suit une des deux stratégies suivantes, choisie en fonction de la stratégie de regroupement de l'objet composite :

- les objets composites représentant les versions alternatives ne partagent pas de pages.

  La transaction demande une nouvelle page (vide d'objets) dans laquelle elle crée les objets. Cette page est utilisée au moment de la validation pour instancier les autres sous-objets de l'objet composite. La création d'objets peut nécessiter plusieurs pages qui ne sont allouées consécutives que dans le cas des objets longs.

- les objets composites représentant les versions alternatives se partagent des pages.

  La transaction instancie les objets créés comme des instances temporaires qu'elle rend aussitôt persistantes (cf. section V.2). Au moment de la validation, les objets temporaires devenus persistants sont recopiés vers les pages partagées par tous les objets composites. Le choix de créer les objets dans la zone temporaire de la transaction évite à la transaction de demander un accès exclusif sur des pages partagées par plusieurs versions et donc de bloquer les autres transactions qui consultent et modifient les autres versions. Cette technique ne bloque ces transactions que durant les validations des transactions. De part de leur nature, les objets longs sont directement créés dans de nouvelles pages.

Les nouveaux objets sont crées avec des identifiants particuliers qui permettront lors de la validation globale d'instancier les objets de la base pour contenir leurs valeurs. Une opération de "unswizzling" sera effectuée sur les objets modifiées dans la base pour convertir ces identifiants particuliers vers les identifiants des objets correspondants dans la base. Dans la figure 5.24, les identifiants particuliers des objets crées sont α', β', χ', δ''. L'objet "table" fait la correspondance entre ces identifiants et ceux d'objets "valeur" qui référencent des pages de la base (la correspondance donne pour α', l'identifiant **a'** d'un objet contenant la valeur **a•** dans V1 alors qu'elle donne l'identifiant **a''** d'un objet contenant la valeur **a••** dans V2).

La destruction d'un objet est indiquée dans une entrée dans la table de correspondance (entrée C dans V1). Cette entrée contient également l'identifiant d'un objet qui permet de rendre la destruction effective lors de l'élaboration de la version définitive si l'objet, référencé par l'entrée, est validé dans cette version. Cet objet correspond en général à l'objet racine d'un objet composite qui sert de grain de coopération de l'application. L'objet composite est susceptible d'être choisi (avec ses sous-objets) au moment de l'élaboration; le choix provoque la destruction effective du sous-objet dans la version définitive.

VII.3.2. Version Modifiée

Quand la version alternative a déjà été modifiée, la spécification retourne l'identifiant de l'objet composite qui représente la version. La transaction consulte et modifie celle-ci en mappant en mode privé les segments qui contiennent les sous objets de la version ou les objets de la base.

La validation de la transaction complète l'objet composite "delta" avec les sous-objets "valeur" qui contiennent les valeurs modifiées des objets de la base ou ceux des versions précédentes. La création d'objets suit les règles vues précédemment. Cependant, dans la stratégie de regroupement, la création d'objets essaie d'instancier les objets dans les pages déjà allouées à la version avant de réclamer de nouvelles pages. Les pages de la version alternative

peuvent être réorganisées afin d'optimiser la structure de l'objet "table". Cette réorganisation peut nécessiter de déplacer des objets "valeur" vers d'autres pages; l'identifiant d'un objet déplacé est alors changé[35]. Ceci est possible car seule la table de correspondance référence ces objets tant que la version n'est pas stable.

VII.3.3. Version Alternatives et Contrôle de Concurrence

L'utilisation de la technique des "deltas" nécessite quelques adaptations du contrôle de la concurrence à l'intérieur de l'Espace de Travail Coopératif et dans ces clients. Le problème réside dans le fait qu'un sous-objet "valeur" peut être référencé par plusieurs objets "delta" si celui-ci n'a pas été modifié depuis une version antérieure (dans la figure 5.24, l'objet "valeur" est partagé par la version stable V1 et la version modifiable V2). Une transaction, qui modifie cet objet partagé à travers une version modifiable, ne doit pas être bloquée tant qu'une autre transaction le consulte ou le modifie à travers une autre version; par la suite, la validation crée un nouvel objet à partir de l'image de l'objet modifié.

Dans le travail coopératif, la concurrence est au niveau des versions alternatives par l'activité superviseur. Le contrôle de concurrence de niveau page, fourni par le fonctionnement standard de l'Espace de Travail, doit être désactivé durant la phase exécutive de la transaction. Dans le cadre de la stratégie de "mélange", le contrôle de concurrence au niveau des pages est utilisé au moment de la validation pour sérialiser les modifications et les créations dans les pages communes.

Cette désactivation interne du contrôle de concurrence ne dispense pas l'Espace de Travail Coopératif de demander au serveur les verrous sur les pages de la base qui ont été consultées ou modifiées[36] par les transactions coopératives ou qui contient des objets de la base présents dans des versions. L'Espace de Travail Coopérative, qui fonctionne suivant le mode englobant, cache les verrous partagés et exclusifs jusqu'à la fin du travail coopératif qui correspond à la validation globale de l'Espace de Travail.

VII.4. Elaboration de la version définitive

Le travail du groupe coopératif se termine par l'élaboration de la version définitive qui consiste en fait à :

- modifier la valeur des objets de la base avec les valeurs des sous-objets "valeur" correspondants, choisis dans les différentes versions alternatives.
- créer de nouveaux objets pour stocker les valeurs des sous-objets "valeurs" créés dans les versions alternatives.
- détruire les objets de la base dont les identifiants sont associés à des objets qui sont modifiés ou créés (cf. section VII.3.1).

L'élaboration de la version définitive est réalisée par l'activité superviseur au moyen d'une transaction ordinaire, à partir des choix d'objets effectués dans les versions alternatives. Le

35 L'identifiant des identifiants des sous-objets "valeur" sont donc réallouables.

36 Les pages pour lesquelles l'Espace de Travail demande des verrous exclusifs, sont celles qui contiennent des objets modifiés dans une des versions alternatives du groupe coopératif.

travail coopératif se termine par la validation globale de l'Espace de Travail qui valident des modifications apportées à la base par l'activité superviseur et relâchent les verrous accordés à l'Espace de Travail. Les versions alternatives associées à ce travail sont abandonnées en simplement remettant les pages, qui les contenaient, dans l'ensemble des pages libres.

VIII. Conclusion

Nous avons décrit dans ce chapitre l'implantation des Espaces de Travail. Cette implantation s'appuie largement sur des mécanismes systèmes récents comme le MultiThreading ou le Memory-Mapping.

Nous avons rappelé dans la section II les principes de ces mécanismes. Nous avons ensuite discuté de l'utilisation du Memory-Mapping dans le cadre d'accès aux données persistantes. Ce choix de conception implique que la structure de l'Espace de Travail utilise la page comme grain de communication, de contrôle de la concurrence.

Dans la section III, nous avons décrit l'accès aux pages Persistantes par les transactions exécutées par des threads d'un Espace de Travail; cet accès rend transparents au code des transactions :

- les fonctions de gestion de la persistance,
- le contrôle de la concurrence,
- et la localisation des données (stockage local ou stockage sur un Espace de Travail distant).

Cependant, l'utilisation du Memory-Mapping comporte quelques inconvénients qui peuvent nuire aux performances de l'Espace de Travail. Nous avons vu qu'il est nécessaire de mettre en place une réplication des données dans les caches des clients. Sa gestion fait l'objet de modifications dans l'implantation de Verrouillage 2-Phases connu sous le nom de Verrouillage par Rappel.

Dans la section IV, nous avons utilisé le MultiThreading pour structurer l'Espace de Travail de façon à permettre un parallélisme entre les transactions applicatives ou de servirce. Ce parallélisme passe par l'utilisation de requêtes asynchrones entre le client et le serveur : nous avons donc ajouté à cette structure des threads déléguées à l'écoute des communications externes pour éviter tout blocage des threads applicatives ou de service. Le MultiThreading est également utilisé pour paralléliser la validation d'une ou plusieurs transactions : un pool de threads est chargé de journaliser les modifications apportées par les transactions. La journalisation parallèle prend en compte les spécificités des threads de service et celles des threads applicatives.

Nous avons structuré dans la section V, les pages persistantes de taille fixe en objets de taille variable. Ces objets sont soit des objets courts que l'on regroupe dans les pages, soit des objets longs constitués de plusieurs pages contiguës dans l'espace d'adressage. Le mode d'instanciation des objets peut être persistant ou temporaire; seul le premier fait l'objet de création effective dans les pages persistantes lors de la validation des transactions. Nous avons expliqué la notion d'identifiant d'objet ainsi que son mécanisme de traduction vers l'adresse d'installation de l'objet à l'intérieur de l'espace d'adressage. Nous avons terminé la section par l'Object Diffing qui permet une validation "objet" du client vers le serveur.

Dans la section VI, nous avons proposé une interface C++ pour l'écriture des transactions exécutées par un Espace de Travail. Cette interface utilise un compilateur du marché et un précompilateur indépendant. Ce dernier redéfinit seulement quelques méthodes dans les classes de l'application et crée des classes références permettant d'accéder aux objets persistants; la déréférenciation qui est normalement réalisé par hachage, peut être améliorée par une technique de swizzling (appelé swizzling "at dereferenciation") quand le client n'effectue qu'une seule transaction à la fois. L'utilisation successive d'un pré-compilateur simplifié et d'un compilateur externe a permis le développement rapide de cette interface sans pour autant perdre les fonctionnalités les plus récentes de ce langage. Néanmoins, nous sommes conscients que ce type d'implantation ne permet pas aux Gérants de faire côtoyer des interfaces différentes sur la même base.

Dans la section VII, nous avons décrit un des principaux mécanismes du travail coopératif. Ce mécanisme est celui des versions alternatives : il permet aux transactions coopératives de consulter successivement plusieurs versions de la base et d'en modifier une parmi elles. La version alternative est implantée selon le principe des "deltas" qui consiste à ne stocker que les valeurs différentes de la version précédente; la version alternative est un objet composite comportant une table de déréférenciation et des sous-objets contenant la valeur des objets modifiés. Cette table introduit dans la déréférenciation des objets un niveau supplémentaire. Cet objet composite disparait en fin de travail coopératif et laisse le résultat de ce travail (i.e. des instances modifiées ou créées) accessible par les applications non coopératives.