Rapport MASI.93.58 Octobre 1993

WEA

Un Gérant d'Objets Persistants basé sur une Hiérarchie Distribuée d'Espaces de Travail

Didier Donsez, Philippe Homond, Pascal Faudemay

Laboratoire MASI / UPMC
4 Place Jussieu, 75252 Paris cedex 05, France
Internet : {donsez, homond, faudemay}@masi.ibp.fr

Résumé

WEA est notre implémentation d'un nouveau modèle d'accès à une grande base d'objets distribuée. Il repose sur la généralisation du modèle client-serveur pour construire aisément les nouvelles applications distribuées. Le WorkSpace fournit un accès uniforme à une mémoire distribuée d'objets persistants. Cet article décrit les différentes façons de combiner les WorkSpaces pour construire des architectures distribuées. L'implémentation du WorkSpace utilise largement des fonctionnalités offertes par les systèmes d'exploitation modernes.

Abstract

WEA is our implementation of a new model for a large distributed object database access, the WorkSpace. It relies on a generalisation of client / server model and enable to build new distributed applications. The WorkSpace supplies a uniform access to a distributed persistent object store This paper describes several ways of building multi-WorkSpace architecture. This implementation is based on new operating systems features

1 Introduction

Les langages à objet représentent une avancée majeure de ces dernières années dans le domaine du Génie Logiciel. Néanmoins, le développeur doit en général utiliser des systèmes de fichiers pour l'archivage des données. Ces derniers ne permettent pas une mémorisation simple des pointeurs sur l'archive, ni la gestion directe des objets sur celle-ci. Les Gérants d'Objets Persistants offrent l'intégration complète et fiable des fonctionnalités des langages de programmation à objets et des systèmes de bases de données.

De nombreux Gérants d'Objets Persistants ont été réalisés [Carey et al 86][Gruber et al 92] et commercialisés [Deux et al 90][Ontologic 90][ATT 93]. La plupart de ces Gérants d'Objets restent cependant inadaptés soit au besoin de nouvelles fonctionnalités , soit au besoin de performance équivalente à celles des langages de programmation classiques.

Parmi les fonctionnalités nécessaires, la distribution complète sur un réseau local suppose l'accès par chaque client à un nombre élevé de serveurs. Le travail de groupe implique la possibilité pour plusieurs transactions de voir les modifications effectuées par chacune d'elles, et de choisir l'une de ces modifications selon une procédure commune au groupe (transactions coopératives). Les applications multimédia doivent gérer des objets contigus de grande taille, accessibles par des mécanismes matériels adaptés (DMA).

Une approche intéressante pour atteindre des performances élevées est d'utiliser pleinement les mécanismes de bas niveau offerts par les systèmes d'exploitation. Cette approche a été commencée avec succès par ObjectStore [Lamb et al. 91]. Ce système gère la correspondance entre les identifiants d'objets [Khoshafian & Copeland 86] et les adresses physiques avec des mécanismes du memory mapping qui permettent d'installer des fichiers dans l'espace d'adressage de l'application.

Les nouveaux systèmes d'exploitation offrent également un accès au parallélisme à travers le MultiThreading [IEEE 92] [SunSoft 93a]. Ce dernier permet à des activités simultanées, les threads, de partager l'espace d'adressage dans l'application. Le MultiThreading permet l'utilisation optimum d'un nombre quelconque de ressources (processeurs, disques, communications, …). Selon nous, ces deux mécanismes doivent servir de point de départ à la conception de nouvelles architectures de Gérants d'Objets.

Notre recherche sur un nouveau type de Gérant d'Objets a commencé par la critique d'un gérant d'objets développé dans le cadre notamment de notre laboratoire, Vroom version 1. Vroom V1 avait pour objet d'implanter une architecture client-serveur performante avec une interface C++, mais n'utilisait pas ces mécanismes systèmes.

Le système WEA (WorkSpace Environnement Architecture) utilise le mapping et le MultiThreading pour implanter un nouveau modèle d'architecture, le modèle WorkSpace (WS

dans la suite de cet article) qui généralise l'architecture client-serveur. Ce modèle est fondé un graphe d'espaces de travail dont chacun peut être simultanément client et serveur. Ce graphe de WorkSpace peut être installé sur un réseau de stations de travail mono ou multiprocesseurs.

La réflexion sur les concepts de base de WEA a fait l'objet d'un travail d'équipe mené au laboratoire MASI; le modèle de graphe d'espace de travail a été formulé par [Abécassis 93]. Dans cet article, nous présentons notre implémentation de ce modèle, le système WEA.

Dans la suite de l'article, nous présentons successivement ce qu'est un espace de travail, les différentes façons d'assembler les WorkSpaces, et enfin les mécanismes internes du WorkSpace.

2 Aspects fonctionnels du WorkSpace

L'espace de travail est un environnement de travail local aux applications à travers lequel elles partagent une base d'objets de façon cohérente et fiable. Avec WEA, nous avons souhaité définir un cadre pour les applications, fondé sur trois modèles. Le premier modèle est basé sur le paradigme de l'objet, le modèle d'exécution est celui d'un multithreading transactionnel et le modèle structurel distribue les objets et les traitements de façon transparente.

Pour cela, nous avons défini une entité, l'espace de travail (WorkSpace). Dans un système de base de données classique, l'espace de travail est l'espace de mémorisation des données locales ou modifiées par la transaction [Eswaran et al. 76]. Ces modifications ne sont rendues visibles aux autres utilisateurs qu'à la fin de transaction, le "commit". Dans l'architecture présentée ici, l'espace de travail correspond également à une vue partielle de la base, contenant des données visibles, directement ou non, par une partie des transactions. L'espace de travail constitue une grande mémoire virtuelle d'objets persistants et temporaires.

2.1 Le WorkSpace comme modèle de programmation objet

Le langage de description et de manipulation est le langage C++ [Ellis & Stroustrup 91]. L'interface de WEA bénéficie de la puissance de ce langage objet qui est devenu un standard de facto sur le marché des bases de données orientées objet. Tout objet C++ est archivable dans une base WEA quelque soit la complexité de sa définition. Le langage est étendu avec des types structurés génériques (tableaux, ensembles, listes et dictionnaires) et avec un type non structuré constitué d'une très longue chaîne d'octets et destiné plus spécialement à l'archivage de données multimédia. Ces types font partie d'une bibliothèque fournie avec l'architecture de WEA.

Dans WEA, les objets sont accédés via leurs identifiants qui sont des références systèmes uniques et non réallouées (sauf dans le cas des objets longs). Certains objets peuvent être nommés par le développeur au moyen de chaînes de caractères pour offrir des points d'entrée dans la base. Par ailleurs, le modèle d'objet est orthogonal à l'instanciation des objets: les objets d'une classe sont indifféremment créés persistants ou bien temporaires. De même, un objet temporaire peut être rendu persistant.

2.2 Le WorkSpace comme modèle transactionnel

Le modèle transactionnel classique s'adapte parfaitement au fonctionnement du WorkSpace. Il est basé sur le découpage des applications en transactions. Chaque transaction encapsule les modifications jusqu'à sa terminaison. A ce moment, l'ensemble des modifications effectuées sont soit toutes validées dans la base, soit toutes abandonnées. Une fois validées, les modifications sont visibles des autres transactions. Pour isoler les modifications, le WorkSpace donne un espace de travail privé à chaque transaction.

Dans un WorkSpace, chaque transaction est exécutée par une thread. Le parallélisme peut être introduit dans une application en instanciant dynamiquement plusieurs transactions simultanées. L'application n'est plus conçue comme une suite séquentielle de consultations et de modifications de la base, mais comme un ensemble d'activités concurrentes.

Les techniques de verrouillage du modèle transactionnel offrent un moyen simple pour synchroniser les accès aux objets par des threads (transactions) concurrentes. Typiquement dans l'exemple d'un outil CASE, l'utilisateur ouvre plusieurs éditeurs pour modifier simultanément des sections de code différentes. Si les éditeurs sont vus comme des transactions par l'outil, la section de code est à accès exclusif et une transaction qui réalise les compilations sera bloquée jusqu'au commit de la dernière modification.

2.3 Le WorkSpace comme modèle structurel

Le WorkSpace offre un accès aux bases objet et structure les applications en transactions. C'est également une entité qui gère la distribution tant au niveau des données qu'au niveau des traitements de façon transparente. Le WorkSpace est **une brique générique** pour construire des applications distribuées.

Les objets de la base sont répartis entre des volumes qui sont placés sur des machines différentes du réseau. Chaque volume est utilisé exclusivement par un seul WorkSpace. Le WorkSpace possède deux grandes caractéristiques: il rend accessible ses volumes privés aux transactions qu'il exécute et il est capable de communiquer avec d'autres WorkSpaces.

Ce mécanisme de communication permet au WorkSpace de rendre public ses volumes privés auprès des autres entités du réseau. Les autres WorkSpaces peuvent demander à s'abonner aux volumes de ce WorkSpace Un dialogue s'instaure pour échanger l'image des pages suivant le mode transactionnel. Ces WorkSpaces sont clients du premier: le WorkSpace unifie les notions classiques de client et de serveur. Le mécanisme de publication-abonnement n'est pas seulement réservé au "service" d'exportation d'images d'objets. Il est mis à la disposition de développeurs qui souhaitent écrire des services suivant le même principe de connexion.

La publication-abonnement n'est pas limitée à un seul niveau: un WorkSpace client peut à son tour publier le service auquel il est abonné ou bien n'importe quel autre. Pour étendre les

possibilités des WorkSpaces, nous avons défini deux types de comportement. Le **passing WS** (WS passant) est un cache de données et de verrous pour les transactions qui s'y exécutent ou pour les WorkSpaces qui y sont abonnés. Le **retaining WS** (WS englobant) se comporte comme une transaction englobante vis à vis des transactions qu'il exécute. Il permet à plusieurs transactions de travailler localement dans le WorkSpace, de partager les modifications qu'elles effectuent sans les valider auprès des volumes (locaux ou distants). Les modifications apportées par ces transactions ne sont validées sur la base qu'au commit du WorkSpace. La combinaison par imbrication de ces deux types de WorkSpace permet de définir et de construire les modèles d'applications que présente la section suivante.

3 La hiérarchie de WorkSpace

Le WorkSpace est une brique qui permet de bâtir des systèmes ou des applications distribuées en s'appuyant sur l'accès à des bases d'objets persistants. Le WorkSpace offre à la fois les fonctions du serveur et celles du client. Cette sous-section présente les différentes façons d'imbriquer les WorkSpaces pour obtenir le modèle classique d'applications client/serveur comme celui plus récent d'applications coopératives distribuées.

3.1 Modèle classique du Client-Serveur.

Le système d'information actuel d'une entreprise est bâtie sur le modèle classique du client/serveur. Dans ce modèle, chaque application exécute une suite de transaction sur une machine cliente et accède à des objets archivés et servis par un certain nombre de machines serveurs. La conception des architectures client-serveur avec le WorkSpace est directe. Trois cas d'utilisation sont envisagés. Le WorkSpace sert une base en publiant un certain nombre de volume et fonctionne en mode serveur (fig.1 WS A). Il peut servir d'intermédiaire face à un WAN et jouer alors un rôle de concentrateur et de cache d'objets pour des clients appartenant au même site (fig.1 WS B). Il sert de support aux clients et exécute les transactions sur des objets importés d'un WS serveurs (fig.1 WS C).

3.2 Modèle de Serveur Applicatif.

Dans un système d'information, certaines données et certains traitements sont trop sensibles pour être traités ou réalisés sur un client. La manipulation des telles données doit être réalisée par un service particulier qui définit une interface contraignant le cadre d'utilisation de ces données sur une machine distincte de celle des clients. Dans ce cas, le WorkSpace est un serveur applicatif (fig.1 WS D) qui publie des services réalisées par des transactions locales. Le programmeur d'un tel service aura le choix d'utiliser le mécanisme de communication (publication, abonnement, dialogue) utilisé pour réaliser le "service d'objet" ou bien d'en définir un autre.

3.3 Modèle Travail de Groupe et Travail Coopératif.

Le travail de groupe vise à faire coopérer plusieurs applications ou personnes dans la réalisation d'un projet commun. Ces applications travaillent sur des parties communes ou distinctes du projet et doivent s'entendre sur une version globale à valider. Ce type de coopération peut être contrôlée par un WorkSpace englobant qui ne valide pas immédiatement les modifications apportées par l'ensemble des transactions ou WorkSpaces clients qui y sont abonnés (fig.1 WS E). Il est nécessaire de définir un mécanisme supplémentaire qui arbitre le choix de la version d'objet définitive pour le groupe.

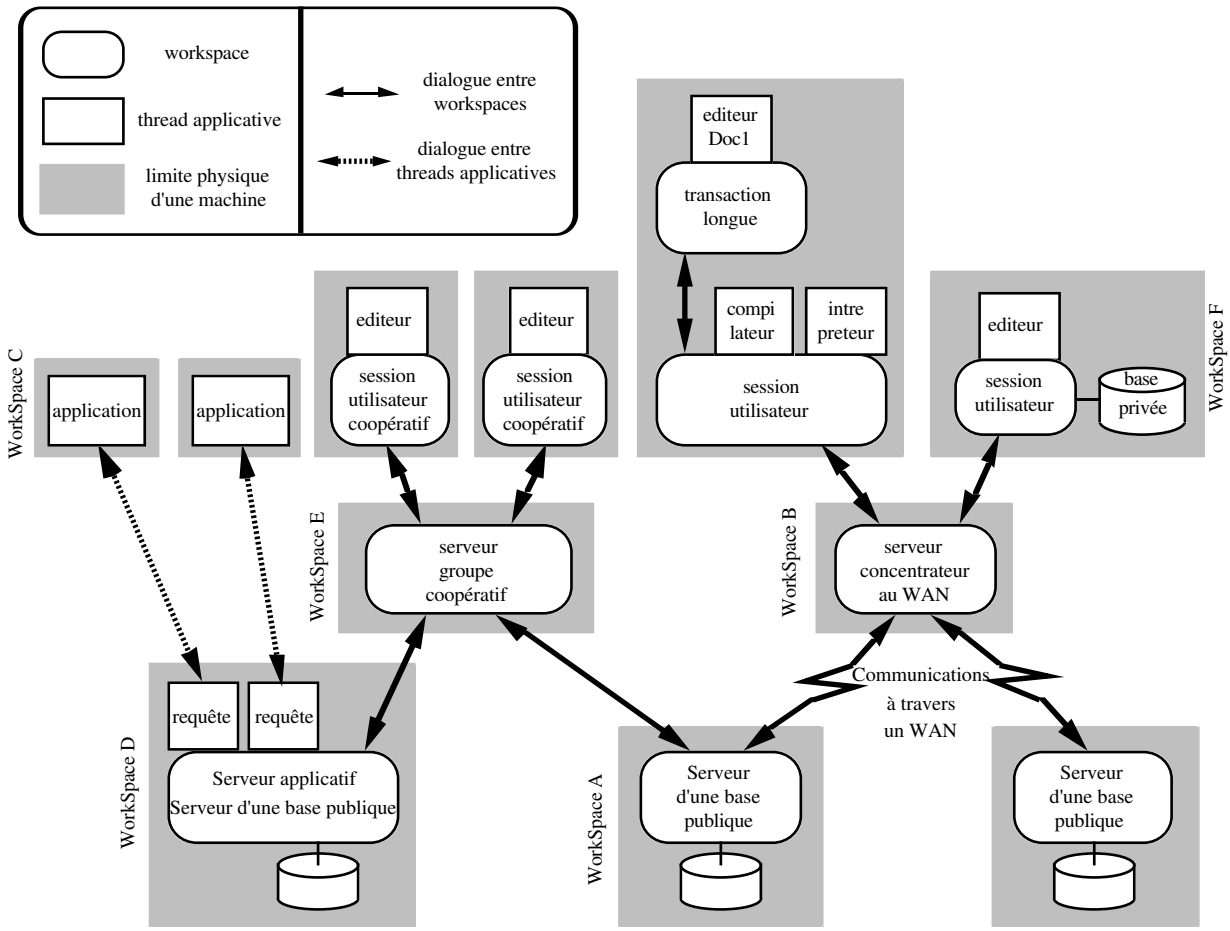

Figure 1 - Composition de WorkSpaces

3.4 Fonctionnement en Stand-Alone ou en Base Privé

L'utilisation de machine stand-alone reste importante chez les particuliers ou les petites entreprises. Un utilisateur doit pouvoir exécuter des transactions sur une base privée (fig.1 WS F) sans être pénalisé par les pseudo-communications toujours nécessaires dans le dialogue entre le client et le serveur. Ce type de fonctionnement est prisé des développeurs qui mettent au point leurs applications sur une base privée de test.

Le code et le binaire écrits restent identiques que la base soit privée ou distante.

3.5 Utilisation Retaining WS / Passing WS.

Ces deux types de WS permettent de construire diverses architectures. Le Retaining WS se comporte comme une transaction englobante et permet d'imbriquer les transactions sur plusieurs niveaux. Cela permet de considérer plusieurs niveaux de travail comme dans le cas du groupeware. Le Passing WS se comporte lui comme un cache de données (objets et verrous) et convient parfaitement à la distribution des données sur un WAN. Il est également utile comme serveur sans transaction locale.

Toute composition de ces deux types de WS est possible ce qui offre une grande flexibilité.

4 Implémentation du WorkSpace

La conception du WorkSpace est largement motivée par les choix techniques d'utiliser le multi-threading et le memory mapping. L'accès à la base d'objet est réalisé par l'intermédiaire de 3 interfaces. L'interface Page utilise le memory mapping pour amener en mémoire une image des pages demandées. Cette interface assure également aux threads un accès transactionnel à ces pages. L'interface Objet introduit la notion d'objet non structuré. Elle distingue deux types d'objet: les objets courts et les objets longs, et définit deux modes d'instanciation: la création d'objets persistants et la création d'objets temporaires. L'interface Langage type les objets des classes persistantes et structure leur contenu. Elle offre l'implémentation standard du langage C++ comme langage de description et de manipulation de ces objets. Le WorkSpace utilise une méthode de contrôle de la concurrence adaptée aux architectures distribuées. Enfin, nous terminerons sur le mécanisme de contrôle d'accès qui utilise ceux d'Unix.

4.1 Structure multithreadée du WorkSpace

Le WorkSpace est un espace d'adressage de mémoire virtuelle (celui d'un processus) que se partagent plusieurs threads. La structure du WS est conçue pour permettre à plusieurs transactions de s'exécuter simultanément , de demander des objets auprès d'un ou plusieurs WS serveurs et de servir des WS clients de façon asynchrone (fig 2):

- La **Local Thread** est le type de thread qui exécute une transaction de l'application. Une application peut instancier plusieurs transactions simultanément. Les transactions profitent ainsi d'un cache commun des pages d'objets chargées dans le WorkSpace.

- Chaque couple de **Threads Mux/DeMux** gère une connexion avec un seul WS serveur. Leur fonctionnement est asynchrone ce qui permet à plusieurs Local Threads d'émettre plusieurs demandes sans sérialiser celles-ci pour attendre les réponses.

- Chaque couple de **Remote Threads** gère une connexion provenant des WorkSpaces clients. La thread RTin est chargée de recevoir les demandes et la thread RTout d'y

répondre. Elles sont la contre-partie des threads Mux/DeMux sur le serveur et fonctionnent comme ces dernières, ce qui rend asynchrone le dialogue entre un client et le serveur.

Le fonctionnement asynchrone du WorkSpace est particulièrement nécessaire dans le cadre d'une hiérarchie de WorkSpace à plusieurs niveaux. En effet, lorsqu'une demande n'est pas satisfaite sur un WorkSpace intermédiaire (§ 3), la demande est transmise au WorkSpace inférieur. Dans l'attente de la réponse, d'autres demandes peuvent arriver et être traitées par le WorkSpace intermédiaire.

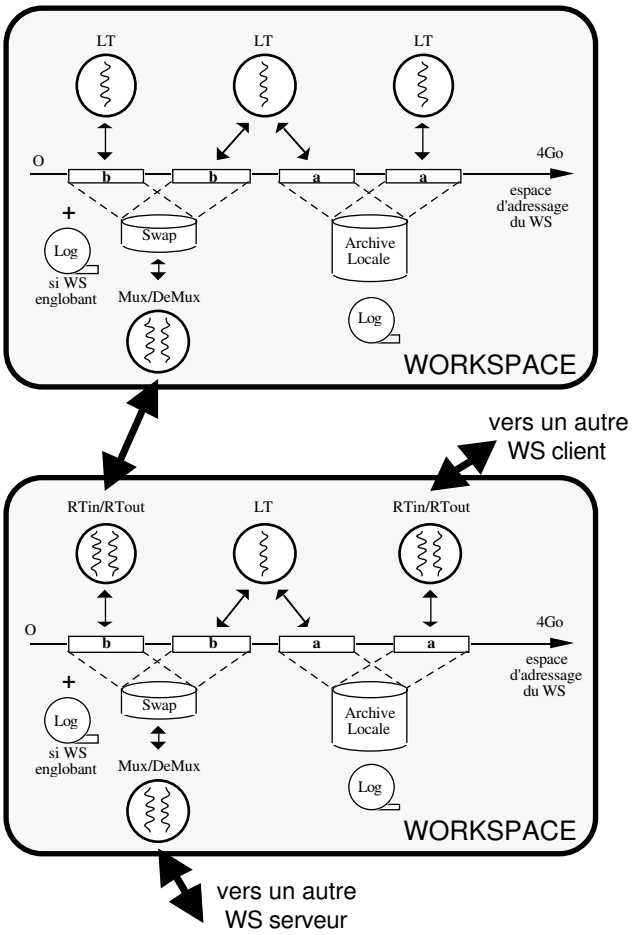

fig. 2 : Fonctionnement interne des WorkSpaces: mapping de segments locaux (a) et distants (b) par les threads .

4.2 L'interface Page

Le rôle de l'interface Page est d'installer dans l'espace d'adressage du WorkSpace l'image des pages de la base demandées par les threads. Les modifications apportées par une thread restent privées à la thread jusqu'au moment de son commit.

Le memory-mapping permet d'installer (on dit "mapper") l'image d'un fichier (ou d'une portion de celui-ci) dans l'espace d'adressage d'un processus [SunSoft 93b]. Les accès au fichier sont remplacés par des accès mémoire dans l'espace d'adressage du processus. Le mapping peut être

partagé et dans ce cas les écritures sont réalisées directement sur le fichier. Le mapping privé préserve lui l'image originale du fichiers en réalisant les écritures sur une copie privée des pages modifiées (le "copy on write"). L'OS permet également de "protéger" des zones de l'espace d'adressage contre des accès en lecture ou en écriture et de détecter ceux-ci quand ils se produisent.

Ces mécanismes sont utilisés pour proposer aux threads du WS une interface d'accès aux pages de la base. Cette interface est transparente par rapport:

- à la localisation des pages qui sont sur des disques locaux ou bien sur les disques de serveurs distants.
- au swap entre les disques et la mémoire primaire.
- au verrouillage des pages

La base est constituée de plusieurs volumes. Un volume accèdé directement est dit local au WS. Ce volume est appelé distant lorsqu'il est accèdé via un WS serveur.

Dans le cas d'un volume local, le memory-mapping est utilisé pour que les threads du WorkSpace puissent mapper directement les volumes (fig 2a). En pratique, comme le volume représente une trop grande quantité de données à mapper d'un seul bloc, le volume est composé de plusieurs segments qui sont des blocs de pages contiguës. Chaque thread qui désire accéder à une page mappe le segment qui la contient. La taille de chaque segment est définie lors de sa création: comme le segment est l'unité de mapping pour les threads, cette taille déterminera la consommation en espace dans la mémoire virtuelle du WS.

Le segment est mappé de façon privée par chaque thread aussi, les modifications apportées ne sont pas recopiées vers les volumes. Ce mapping permet néanmoins de partager les pages non modifiées. Au commit de la thread, les pages modifiées dans le mapping privé sont recopiées vers le segment via un mapping partagé de celui-ci[1]. L'image précédente de chaque page modifiée est auparavant recopiée dans un journal dit des "images avant". Ce journal permet de défaire les threads qui échoueraient pendant leur commit.

Le verrouillage des pages est détecté par le mécanisme de protection de la mémoire virtuelle. Initialement, toutes les pages d'un segment mappé par une thread sont protégées contre les accès en lecture et en écriture. Par la suite, chaque nouvel accès dans une page du segment lève une exception signalant la violation des protections (signal Unix `SIGSEGV`). La demande de verrou (en lecture et/ou en écriture) est alors effectuée. Une fois le verrou obtenu (soit immédiatement, soit quand celui-ci a été relâché par la ou les threads qui le possèdaient), la thread est relancée à partir de l'instruction fautive.

---

[1] Une optimisation permet aux Remotes Threads d'utiliser ce mapping partagé pour accéder aux pages car elles ne modifient ces dernieres que durant les phases de commit.

Dans le cas d'un volume distant, un dialogue s'établit entre le WS client et le WS serveur pour l'importation des pages demandées à travers le réseau (fig 2b). Les pages d'un segment distant ainsi importées sont copiées dans un fichier local qui joue le rôle d'image locale. Ce fichier image est alors mappé par les threads du WS qui désirent accéder aux pages du segment distant. Cette image est incomplète et lorsque l'image d'une page n'est pas présente dans le fichier, la demande de cette page (et du verrou associé) est envoyée au WS serveur. La thread est bloquée[2] jusqu'au retour de la page. L'utilisation du mapping privé sur le fichier image et la détection des demandes de verrous sont identiques à celles d'un segment local. Le mapping de ce fichier image joue le rôle de swap des pages des volumes distants.

Le commit de la thread diffère suivant le mode de fonctionnement du WS auquel elle est attachée:

- Dans un WS passant, les pages modifiées par chaque thread sont recopiées vers le fichier image et sont également renvoyés vers le WS serveur du volume.
- Dans le cas d'un fonctionnement en mode Englobant (Retaining WS), les pages modifiées sont seulement recopiées vers le fichier image après avoir été journalisée localement. Au moment de son "commit", le WorkSpace englobant retourne au serveur l'ensemble des pages modifiées par les threads commises. Comme nous le verrons dans l'alinéa sur le contrôle de concurrence, cette phase correspond aussi au relâchement de verrous en écriture.

Le memory-mapping permet d'installer théoriquement jusqu'à 4Go d'objets de la base dans la mémoire virtuelle d'un WS durant la "vie" de celui-ci. Cette limite est pénalisante pour les WS serveurs qui en principe ont une durée de service assez longue. Aussi, un mécanisme de récupération de l'espace dans la mémoire virtuelle désalloue les mapping de certains segments pour les attribuer à d'autres. Il faut remarquer que les mappings privés des segments utilisés par des Local Threads ne peuvent pas être swappés.

4.3 L'interface Objet

L'interface Page décrite à l'alinéa précédent ne connait pas la structure interne des pages. Elle est complétée par l'interface Objet qui structure le contenu des pages en listes d'objets. L'interface objet permet la création et la destruction d'objets courts ou longs, persistants ou temporaires et assure la déréférenciation vers ceux-ci.

4.3.1 Objet court, objet long.

Cette interface manipule deux types d'objets non structurés: les objets courts et les objets longs. Les objets courts sont destinés à accueillir les instances des classes C++ que l'interface Langage définit. L'objet court contiendra le numéro de la classe à laquelle il appartient. Ce numéro servira à

---

[2] Cette opération n'est pas bloquante pour les Remote Threads qui sont en fait en attente de n'importe quel retour d'une demande émise.

l'interface langage pour rafraîchir le pointeur mémoire vers la table des méthodes virtuelles. Ce pointeur n'est valide que pour un WorkSpace, aussi il est mis à jour (on dit aussi "rafraîchi") à chaque fois que l'objet est chargé dans un nouveau WorkSpace. L'objet court peut contenir également des références persistantes vers d'autres objets. La taille de l'objet court ne doit pas dépasser la taille d'une page.

L'objet long est un long tableau d'octets chargé dans un ensemble de pages contigües en mémoire virtuelle. Les objets longs sont concus pour archiver des données multi-média telles que des images ou des sons. La contiguité des pages en mémoire assure la manipulation de ces objets par des accélérateurs matériels.

Un objet (court ou long) est instancié soit de façon permanente (persistante), soit de façon temporaire. Un objet créé permanent est réellement alloué dans la base au commit de la transaction alors qu'au même moment, l'objet créé temporaire disparait. Les objets temporaires n'existent que le temps de la transaction. Néanmoins, une méthode permet de rendre permanent un objet créé temporaire et de "survivre" au commit.

Lors de sa création, un objet persistant peut être placé "à coté" d'un autre objet déjà existant, dans la même page. Ce placement des objets à la création optimise les accès disque et réseau quand ces deux objets sont utilisées simultanément.

4.3.2 Identifiant d'objet et Déréférenciation

Un objet est accessible par son identifiant. L'identifiant d'objet est une référence unique dans l'ensemble du système [Khoshafian & Copeland 1986]. 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 référençent des objets détruits. Une référence vers un objet détruit retourne une valeur nulle.

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 discuté dans l'alinéa précédent, 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 le WS 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.

La déréférenciation est l'opération de traduction de l'identifiant d'un objet vers l'adresse mémoire à laquelle est installée une image de cet objet. Cette opération, jugée critique pour les systèmes d'objets persistants, ne nécessite dans WEA qu'une indirection effective qui donne

l'adresse de mapping du segment. Cette indirection est faite via une table de hachage de petite taille résidant donc en mémoire. Il existe une deuxième indirection à l'intérieur de la page qui donne la position relative d'un objet dans la page. Elle est faite au moyen d'un tableau situé dans l'entête de page et qui est donc chargé en mémoire en même temps que l'objet. Cette dernière indirection ne nécessite guère que quelques instructions.

La taille de l'identifiant est un des paramètres d'instanciation 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 avec 4To d'objets crées dans la vie de la base (pour une base vide d'objets longs). La base d'objets longs est d'au maximum de 4 To (pour une base vide d'objets courts). Le codage sur 64 bits offre une taille de base quasi-illimité mais augmente la taille des références persistantes dans les objets.

4.3.3 Stratégies de création d'objets

Les objets temporaires sont alloués dans une zone temporaire privée de la thread. Au commit de la thread, les objets temporaires devenus persistants sont recopiés de sa zone temporaire vers les pages que la thread a déjà verrouillées en écriture. S'il n'y a pas assez de places dans ces pages, la thread demande un intervalle de pages libres pour y placer les objets restants. Les objets temporaires désormais persistants se voient attribués un nouvel identifiant dans les pages persistantes. Il est nécessaire de remettre à jour les références qui peuvent exister vers ces objets: c'est à dire dans les objets nouvellement créés ou modifiés par la thread.

Les objets persistants sont crées dans les pages verrouillées en écriture par la thread. S'il n'y a plus de place, le WorkSpace choisit entre 2 stratégies de création. La première stratégie demande au fur et à mesure de nouvelles pages pour y créer ses objets. Le seconde stratégie préfère alors ses objets créés dans sa zone temporaire et attendre le commit pour les placer dans les pages verrouillées en écriture aprés la création. Cette stratégie favorise le remplissage des pages par rapport à la première mais nécessite un recopie mémoire des objets. Le WorkSpace bufferise un intervalle de pages libres par rapport aux volumes distants.

La création d'un objet à proximité d'un objet existant correspond au placement des deux objets dans la même page (clustering). Dans ce cas, le verrou en écriture de la page est systématiquement demandé par la transaction. Cette demande n'entraîne pas forcement un surcoût de verrouillage: l'objet créé est souvent référencé par l'objet existant ce qui nécessite de modifier cet objet existant et donc d'obtenir le verrou en écriture sur la page.

4.4 L'interface Langage: le typage des objets.

L'interface Langage type les objets des classes persistantes et structure leur contenu. Elle offre l'implémentation standard du langage C++ comme langage de description et de manipulation de ces objets. L'interface Langage répertorie l'ensemble des classes C++ utilisées par les WorkSpaces et d'instancier ces classes dans les objets courts définis par l'interface Objet .

4.4.1 Objets et Outils de l'interface

Les outils de cette interface se compose d'un parser et d'un compilateur C++ quelconque. Le parser répertorie les classes C++ persistantes en les ajoutant à la métabase. L'analyse de la définition de chacune de ces classes est suffisante pour générer le code des méthodes nécessaires pour créer, détruire, "rafraichir" ou référencer les objets.

Le parser utilise les fonctionnalités de surcharge d'opérateur (overloading) du langage C++. Ainsi l'opérateur `new()` de chaque classe persistante est redéfini pour créer les objets (temporaires ou persistantes). L'exemple décrit dans l'alinéa 4.4.2 procède à plusieurs créations d'objets au cours d'une transaction. Pour chaque classe, le parser utilise également le constructeur pour remettre à jour ("rafraichir") le pointeur vers la table de méthodes virtuelles de la classe pour les objets nouvellement chargés dans le WorkSpace. Le "rafraichissement" est possible car chaque objet défini dans l'interface possède le numéro de la classe à laquelle il appartient. La même technique est utilisée par appeler les méthodes nécessaires au commit des objets des pages modifiées.

Pour chaque classe, le parser définit un type pointeur persistant vers la classe. Ce type est classe dont l'opérateur `->()` est surchargé pour appeler l'opération de déréférenciation (§4.3.2). Cet opérateur référence indifféremment les instances persistantes ou temporaires de la classe.

Le compilateur C++ est un compilateur quelconque du marché (ATT ou GNU). Il génère le code des applications à partir de code défini par le développeur et à partir des méthodes générées par le parser. Cette manière de procéder à plusieurs avantages. Elle 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.

Le langage C++ est complétée par une bibliothèque de collections génériques d'objets persistants. Ces collections sont les listes, les ensembles, les tableaux dynamiques et les dictionnaires. Elles viennent en annexe par respect pour la norme C++. Ces collections peuvent être intégrées dans la définition d'une classe persistante..

4.4.2 Exemple de transaction

Nous considérons dans cet exemple le schéma suivant, définissant une classe `Person`. La définition de cette classe respecte la syntaxe du C++ pur. Dans cette classe, le tableau `parents` référence des objets persistants (désignés par le nom de la classe préfixée de `p_`). De même, le tableau dynamique `children` référence d'autres instances de cette classe `Person`. La classe posséde deux constructeurs définis par l'utilisateur. Ces constructeurs n'entrent pas en conflit avec les constructeurs par défaut défini par le parser.

définition du schéma

```
class Person { public:
    char              name[64];
    int               age;
    p_Person          parents[2];       // refs to persitent object
    p_Seq(Person)     children;         // array of persitent object

    Person () { age = 0; }
    Person (char* name) { this->name = name; }
};
```

L'application ci-dessous définit une transaction qui va créer des objets persistants.

exemple de transaction

```
WEA::connect("Bible");           // connexion à la base Bible

W_Transaction *transaction = WEA::newTrans();
    // début de transaction

p_Person eve  = new              Person("Eve");
p_Person adam = new(W_PERS)      Person("Adam");
p_Person abel = new(W_TEMP)      Person;  // création d'objet temporaire

adam->children += new(adam)      Person("Caïn"); // création à proximité
…
abel->name           = "Abel";
abel->age            = 20;
abel->parents[0]     = adam;
abel->parents[1]     = eve;
…
abel->persistent();              // on rend persistant l'objet abel
abel->setName("ABEL");           // abel possède un nom externe
…
transaction->commit();     // fin de transaction
WEA::disconnect();               // déconnexion de la base
```

La classe `WEA` est le point d'entrée pour utiliser les fonctions du WorkSpace. Elle permet de connecter / déconnecter le Workspace aux bases et de démarrer de nouvelles transactions. Dans cet exemple, l'unique transaction crée 4 instances de la classe `Person` qui sont désignées par les 4 pointeurs persistants Les instances designées par `eve` et `adam` sont créées persitantes (avec appel des constructeurs définis dans le schéma) alors que celle désignée par `abel` est créé temporaire. L'instance designée par `caïn` est crée persistante à proximité de l'instance désignée par `adam` (§4.3.1). La surcharge de l'opérateur ->() permet de référencer les objets indifféremment créés persistant ou temporaires. La méthode `persistent()` permet de rendre persistant un objet créé temporaire (`abel` dans cet exemple). La méthode `setName()` permet d'associer un nom externe à un objet. Cet objet pourra alors être retrouvé dans la base à partir de son nom externe. La fin de transaction est indiquée par `commit()` (ce qui valide dans le base les modifications) ou par `abort()` (ce qui abandonne les modifications). Après le commit, les objets créés sont recopiés dans la base et deviennent accessibles par d'autres transactions.

dans une autre transaction

```
WEA->connect("Bible");
W_Transaction *transaction = WEA->newTrans();

p_Person person("ABEL");

person->parent[1]->age++; // verrouillage implicite en ecriture de "eve"

transaction->commit();    // ou WEA->abort();
WEA->disconnect();
```

Dans cette transaction, l'instance précédenment désigné par `abel` est retrouvé dans la base grace à son nom externe `"ABEL"`. L'age de sa mère (`eve`) est incrémenté. Cette ligne a nécessité le verrouillage en lecture de l'instance désignée par `person` et le verrouillage en écriture de l'instance désignée par `person->parent[1]`. Ces 2 verrouillages sont implicites. Le contrôle de concurrence présenté dans la suite, bloque les autres transactions cherchant à accéder à `"eve"` jusqu'au commit de notre transaction.

4.5 Les fonctions Bases de Données.

4.5.1 Contrôle de Concurrence: le Callback Locking hiérarchique.

[Wang & Rowe 91] présentent et comparent plusieurs méthodes de contrôle de concurrence pour des architectures Bases de Données client/serveur. Les résultats de leur simulation mettent en relief la méthode du **Callback Locking** (CL). Le CL est basé sur le verrouillage 2-phase (TPL Two-Phase Locking) mais maintient sur le client un cache des verrous en lecture obtenus par une des transactions qui s'y exécutent. La demande du verrou en écriture nécessite par contre

d'invalider le verrou en lecture dans le cache de chaque client. Ce verrou en lecture est relâché par le WorkSpace client quand il est réclamé et lorsque toutes ses transactions lectrices ont commis. Le WorkSpace serveur sert à centraliser et diffuser les demandes de verrous. Cette méthode est particulièrement adaptée aux situations où les transactions d'un même client référencent les mêmes objets. C'est dans ce type de situation que se placent généralement les applications qui utilisent les WorkSpaces.

Le Callback Locking a été adaptée à l'imbrication hiérarchique de l'architecture en WorkSpace: chaque niveau d'imbrication devient serveur de verrous pour le niveau supérieur. Néanmoins, le choix est laissé au WS d'inhiber le cache des verrous en lecture pour revenir localement à un fonctionnement similaire aux implantations distribuées du TPL classique.

Le mode de fonctionnement du WorkSpace influe également sur le relâchement des verrous en écriture. Le WorkSpace passant relâche les verrous en écriture après le commit de chaque thread alors que le WorkSpace englobant les conservent jusqu'à son propre commit.

Le grain de verrouillage est la page. Ce grain permet des détecter les demandes de verrou par les mécanismes de protection de la mémoire virtuelle ce que n'est pas possible avec le grain Objet. Le WorkSpace n'a pas une connaissance précise sur l'état des verrouillages posés par les threads de ses clients. Ceci ne permet pas de dresser un graphe des dépendances entre les threads pour détecter les interblocages. Les interblocages possibles sont détectés par un timeout qui lève une exception dans la thread (mécanisme standardisé dans le C++ 4.0 d'ATT). Les problèmes d'optimisation des timeouts ont été étudiées dans la machine Gamma [DeWitt et al 1990].

4.5.2 Fiabilité et Reprise sur Panne

La reprise sur panne est réalisée de façon classique avec un mécanisme de journal des images avant. Sur un WorkSpace englobant, le journal local sert également à défaire les threads qui abandonnent pendant leur commit les modifications faites sur un volume distant. Dans une version future de WEA, ces images avant pourront être également consultées pour extraire des versions historiques d'objets.

4.5.3 Sécurité et droits d'accès

WEA utilise les mécanismes de droits d'accès et d'identification d'Unix. Le WorkSpace possède l'identifiant d'utilisateur (UID) et de groupe (GID) de l'utilisateur qui le demarre. Dans un volume, les objets possèdent tous ses droits d'accès (propriétaire, groupe, autre). Ce niveau de contrôle est lié au fait qu'un volume est stocké dans un fichier unix ce qui n'introduit pas de trou dans la sécurité du système. Dans une version future, le contrôle d'accès pourra être descendu au niveau du segment ou de la page pour les volumes servis par des Workspace distants contrôlant les droits lors de chaque accès.

5 Conclusion

WEA est une brique générique pour concevoir les différents modèles d'applications distribuées (client-serveur classique, serveur applicatif, …). Il offre aux applications un accès transparent à une grande mémoire virtuelle d'objets distribuée sur un réseau de station de travail. Son interface C++ lui permet de bénéficier de la puissance de ce langage. La conception interne des WorkSpaces utilise les techniques avancées proposées par les nouveaux systèmes d'exploitation. WEA se situe à cheval entre le système et le langage ce qui lui permet de profiter pleinement des avancées technologiques tout en restant indépendant.

La conception de WEA a commencé en Octobre 1992 et un premier prototype sera terminé en Décembre 1993. Les futurs travaux consisteront à étendre le WorkSpace pour supporter le travail coopératif et les versions. Cette extension passe par la spécialisation du modèle de donnée et du modèle transactionnel.

6 Bibliographie

[Abécassis 93] E. Abécassis, communication privée, 1993

[ATT 93] ATT Bell Laboratory, "ODE 2.0 User's Manual", 1993

[Deux et al 90] O. Deux et al, "The Story of O2", IEEE transactions on knowledge and data engineering vol.2 n°1 1990

[DeWitt et al. 90] David J. DeWitt, Shahram Ghanderarizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao and Rick Rasmussen, "The Gamma Database Machine Project",IEEE Transactions on Knowledge ans Data Engineering, Vol 2, No 1, March 1990, p44-62

[Ellis & Stroustrup 91] M.A. Ellis, B. Stroustrup, "The Annoted C++ Reference Manual", Addison-Wesley, 1991

[Eswaran et al. 76] Eswaran K.P., Gray J.N., Lorie R., Traiger L.L., "The Notion of Consistency and Predicate Locks in a Database System", Comm. ACM, 19, 11, Nov.1976

[Gruber et al 92] O. Gruber and L. Amsaleg and L. Daynes and P. Valduriez, "EOS, An Environment for Object-Based Systems",Proc. of the 25th Hawaii International Conference on System Sciences", jan 1992, Vol 1, pp757-768

[IEEE 92] IEEE, "POSIX 1003.4a Draft 6 - Threads Extension for Portable Operating Systems", Feb 1992.

[Khoshafian & Copeland 86] S. Khoshafian and G. Copeland, "Object Identity", MCC Research Report, 1986.

[Lamb et al. 91] Charles Lamb, Gordon Landis, Jack Orenstein, Don Weinreb, "The ObjectStore Database System", Communication of the ACM October 1991, Vol. 34, No. 10, pp50-63

[Ontologic 90] Ontologic Inc. "ONTOS Object Database Programmer's Guide", February 1990.

[SunSoft 93a] SunSoft, SunOS 5.2 Guide to MultiThread Programming, Answer Book, March 1993

[SunSoft 93b] SunSoft, SunOS 5.2 Memory Management, Answer Book, March 1993.

[Wang & Rowe 91] Yongdong Wang, Lawrence A. Rowe, "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture", SIGMOD'91, pp367-376

7 Table des Matiéres

1 Introduction 2
2 Aspects fonctionnels du WorkSpace 3
2.1 Le WorkSpace comme modèle de programmation objet 3
2.2 Le WorkSpace comme modèle transactionnel 4
2.3 Le WorkSpace comme modèle structurel 4
3 La hiérarchie de WorkSpace 5
3.1 Modèle classique du Client-Serveur. 5
3.2 Modèle de Serveur Applicatif. 5
3.3 Modèle Travail de Groupe et Travail Coopératif. 6
3.4 Fonctionnement en Stand-Alone ou en Base Privé 6
3.5 Utilisation Retaining WS / Passing WS. 7
4 Implémentation du WorkSpace 7
4.1 Structure multithreadée du WorkSpace 7
4.2 L'interface Page 8
4.3 L'interface Objet 10
4.3.1 Objet court, objet long. 10
4.3.2 Identifiant d'objet et Déréférenciation 11
4.3.3 Stratégies de création d'objets 12
4.4 L'interface Langage: le typage des objets. 13
4.4.1 Objets et Outils de l'interface 13
4.4.2 Exemple de transaction 14
4.5 Les fonctions Bases de Données. 15
4.5.1 Contrôle de Concurrence: le Callback Locking hiérarchique. 15
4.5.2 Fiabilité et Reprise sur Panne 16
4.5.3 Sécurité et droits d'accès 16
5 Conclusion 17
6 Bibliographie 17
7 Table des Matiéres 18