WEA,
Des Espaces de Travail Distribués
à Objets Persistants

Didier Donsez, Philippe Homond

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

Résumé

WEA est un gérant d'objets distribués interfacé au langage C++. Son architecture est basée sur un graphe de Workspaces. Il généralise le modèle client / serveur en permettant la construction d'applications distribuées. Le WorkSpace fournit un accès uniforme à une mémoire distribuée d'objets persistants. Cet article décrit les différentes façons de concevoir des applications autour d'une hiérarchie de WorkSpaces ainsi que l'implantation de ceux-ci. Cette implantation repose sur l'utilisation du multithreading et du memory-mapping.

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 et commercialisés [Deux90] [ATT93]. La plupart de ces Gérants d'Objets restent cependant inadaptés soit aux besoins des nouvelles applications telles que le travail coopératif et le multimédia, soit à l'obtention de performances équivalentes à celle des langages de programmation classiques. Le travail coopératif implique la possibilité pour plusieurs transactions de voir les modifications effectuées par les autres et de choisir l'une de ces modifications. 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 [Lamb91]. Ce système utilise le memory-mapping pour accéder aux objets directement dans la base. Selon nous, cette approche doit être complétée par l'utilisation du MultiThreading [Sun93a] qui permet d'introduire du parallélisme au sein des applications.

Actes des *Journées des Jeunes Chercheurs en Systèmes à Mémoire Logiquement Partagée,* Toulouse, Septembre 1993.

Le système WEA utilise ce deux techniques pour implanter un nouveau modèle d'architecture (WS dans la suite de cet article) qui généralise l'architecture client-serveur. Ce modèle est fondé sur un graphe d'espaces de travail dont chacun peut être simultanément client et serveur.

Dans la suite de l'article, nous présentons successivement les fonctionnalités d'un WS, les différentes façons de les assembler et enfin ses mécanismes internes.

2. Aspects fonctionnels du WorkSpace

Le modèle d'architecture en WorkSpace est destiné à bâtir des applications distribuées en connectant des entités remplissant à la fois la fonction de client et celle de serveur. Le WorkSpace est un environnement de travail local aux applications à travers lequel elles partagent une base d'objets de façon cohérente et fiable. Les fonctionnalités du WS s'orientent par rapport aux 4 axes suivants:

- Mémoire Virtuelle d'objets
- Mode d'exécution transactionnel
- Accès transparent à une base d'objets distribuée.
- Communication entre WorkSpaces.

Le WS offre aux applications une grande mémoire virtuelle d'objets persistants ou temporaires. Cet espace mémoire est à la fois la zone de travail où les applications stockent les objets auxquels elles accèdent et une image partielle de la base sous-jascente. Le langage de description et de manipulation des objets est le C++. Tout objet C++ est archivable dans une base WEA quel que soit la complexité de sa définition. WEA étend ce modèle objet en introduisant un type non structuré, l'objet long, destiné à l'archivage des données multimédia. Les objets de la base sont accédés via des références systèmes uniques et non réallouées.

Le mode d'exécution de WEA est transactionnel. Une application est alors un ensemble de transactions concurrentes qui consultent et modifient la base. Chaque transaction encapsule les modifications jusqu'à sa terminaison, puis les valide dans la base pour les rendre visibles des autres transactions [Garz88]. Une transaction dans WEA est exécutée par une thread. Ce mode d'exécution couplé à un mécanisme de verrouillage (le callback locking dans WEA) assure un accès cohérent à la base en synchronisant les accès aux objets.

Le WorkSpace offre un accès aux bases d'objets et structure les applications en transactions. Une base WEA est constituée de volumes répartis sur différentes machines du réseau, chacun étant utilisé par un unique WorkSpace. Un mécanisme de publication / abonnement permet aux WorkSpaces de rendre public l'accès à un volume ou à tout autre service. Un WorkSpace abonné à un autre devient client de ce dernier et peut alors en obtenir les services publiés. Le WorkSpace muni de cette méthode de communication est une brique générique pour construire des applications distribuées. Les WorkSpaces abonnés entre eux permettent d'accéder de façon transparente à une base distribuée. Pour étendre les combinaisons possibles entre WS, nous avons défini deux types de comportement. Le **passing WS** (WS passant) fonctionne comme un cache de données et de verrous pour les transactions qui s'y exécutent et pour les WS 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 WS et de partager leurs modifications sans les valider dans la base. L'imbrication de WS permet de construire différents modèles d'applications que présente la section suivante.

3. La hiérarchie de WorkSpaces

L'imbrication de WorkSpaces permet d'obtenir le modèle classique d'application client / serveur comme celui plus récent d'application coopérative distribuée. La structure du WorkSpace lui permet de servir une base publique (fig.1 WS A) ou de jouer le rôle d'intermédiaire face à un WAN. Il sert également de support aux clients et exécute les transactions sur des objets importés des WS serveurs (fig.1 WS C). Une application peut publier des services et ainsi définir une interface contraignant le cadre d'utilisation de ses données. Les services sont exécutés par des transactions locales au serveur applicatif (fig.1 WS D). D'autre part, un WS englobant peut gérer le travail coopératif entre plusieurs applications travaillant sur des parties communes d'un même projet (fig.1 WS E). Ce WS organise les décisions et valide une version du projet. La distinction WS englobant / passant permet de construire de nombreuses architectures. L'accès à une base distribuée est transparent avec le WS. Le code d'une application est identique que la base soit locale ou distante.

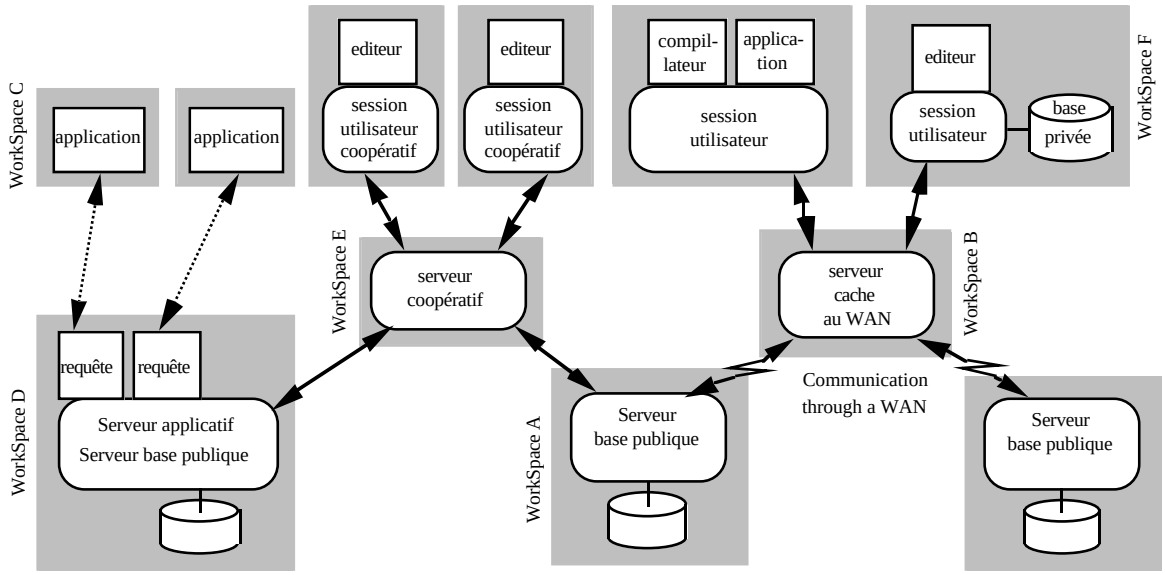

Figure 1 - Composition de Workspaces

4. Conception Interne du WorkSpace

La conception du WorkSpace est largement motivée par les choix techniques d'utiliser le MultiThreading et le Memory-Mapping. L'accès à la base d'objets est réalisé par l'intermédiaire de 3 interfaces: l'interface Page assure aux threads un accès transactionnel aux pages de la base, l'interface Objet introduit la notion d'objet non structuré et l'interface Langage offre le C++ comme langage de description et de manipulation des objets persistants. Notre méthode de contrôle de la concurrence favorise l'utilisation successive des verrous demandés.

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 exécuter simultanément plusieurs transactions, pour demander des objets auprès d'un ou plusieurs WS serveurs ou pour servir des WS clients de façon asynchrone.

Dans le WorkSpace (fig. 2), les transactions sont exécutées par les **Local Threads**. Les transactions profitent ainsi d'un cache commun de pages d'objets chargées dans le WorkSpace. Les autres threads, invisibles à l'utilisateur, gèrent les publications et les abonnements entre les WorkSpaces. Chaque abonnement est géré par un couple de **Threads Mux/DeMux** sur le client et par un couple de **Remote Threads** sur le serveur. Leur fonctionnement est asynchrone, ce qui permet à plusieurs threads du client (Local ou Remote Threads) d'émettre plusieurs demandes sans sérialiser celles-ci pour attendre les réponses. Ce fonctionnement est particulièrement nécessaire dans le cadre d'une hiérarchie de WorkSpaces à plusieurs niveaux: une demande émise du client ne doit pas bloquer les WorkSpaces intermédiaires jusqu'au WorkSpace serveur.

L'interface Page

L'interface Page installe dans l'espace d'adressage du WorkSpace l'image des pages de la base demandées par les threads. Le Memory-Mapping rend ces demandes transparentes par rapport à la distribution des pages sur le réseau. Le verrouillage implicite des pages est effectué par les mécanismes de protection de la mémoire.

La base est répartie entre plusieurs **volumes**. Chaque volume se compose de plusieurs blocs de pages contiguës appelés **segments**. Le segment est l'unité de mapping dans le WS: quand une thread désire accéder à une ou plusieurs pages d'un segment, elle installe ("mappe") celui-ci dans l'espace d'adressage du WS. La taille de chaque segment est définie lors de sa création.

Le segment est dit local lorsque le WS y accède directement et distant lorsqu'il y accède par l'intermédiaire d'un autre WS. Un segment local est mappé directement par la thread. Pour accéder à un segment distant, la thread mappe un fichier local constitué d'une image partielle du segment distant. Cette image partielle contient les pages chargées depuis le serveur par le couple de threads Mux/DeMux au fur et à mesure des demandes.

Le segment local est mappé en mode privé (fig. 2a): pour chaque thread, les modifications restent privées jusqu'à son commit. Au commit, les pages modifiées sont recopiées vers le segment via un mappage partagé de celui-ci. 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 fichier image d'un segment distant est également mappé en mode privé (fig. 2b). Le commit recopie les pages modifiées vers ce fichier et retourne celles-ci vers le serveur au travers du couple Mux/DeMux.

Le verrouillage implicite des pages est réalisé avec le mécanisme de protection de la mémoire virtuelle. Initialement, le mapping d'une thread est protégé contre ses propres accès en lecture et en écriture. Par la suite, chaque nouvel accès dans une page du mapping provoque une exception qui est interprété comme une demande de verrou sur la page. Une fois le verrou obtenu (soit immédiatement, soit quand le verrou est relâché par la ou les threads qui le possèdent), la thread est relancée à partir du point d'interruption.

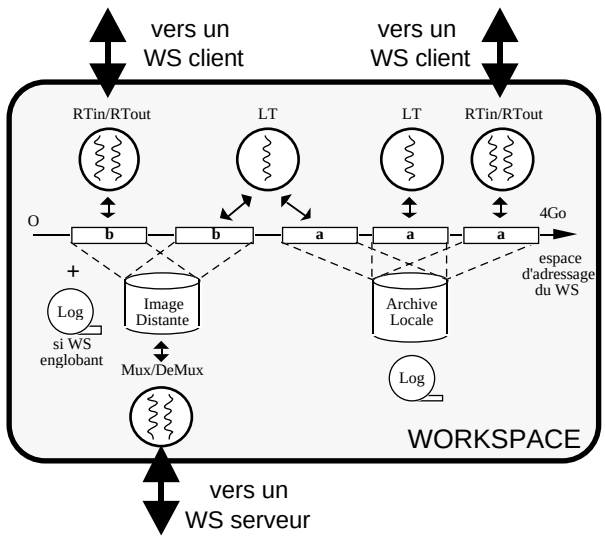

fig 2: mapping des segments locaux (a) et distants (b) par les threads .

L'interface Objet

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++ définies par l'interface Langage. L'objet court contient l'identifiant de sa classe et le pointeur mémoire vers la table des méthodes virtuelles de sa classe. Ce pointeur est mis à jour ("rafraîchi") chaque fois que l'objet est chargé dans un nouveau WS. L'objet long est destiné à archiver un seul type de données: les tableaux d'octets. L'objet long est conçu spécialement pour archiver des données multimédia telles que des images ou des sons. Il est chargé en mémoire virtuelle dans un ensemble de pages contiguës pour pouvoir manipuler ce type de données au moyen de composants matériels spécialisés.

Les objets sont instanciés permanents ou temporaires selon qu'ils persistent ou non après le commit de la transaction. Néanmoins, une méthode permet de rendre permanent un objet créé temporaire. Tout objet persistant peut être placé "à coté" d'un autre objet déjà existant. Ce placement des objets à la création optimise les accès disque et réseau quand les deux objets sont utilisés simultanément.

L'identifiant d'un objet temporaire est une adresse dans l'espace d'adressage alors que l'identifiant d'un objet court persistant se compose des numéros du volume, du segment et de la page dans lesquels il a été crée et ainsi que d'un numéro d'objet relatif à la page. L'identifiant d'objet long se compose des numéros du volume, du segment et de la page où il commence. La traduction de l'identifiant d'un objet en adresse mémoire ne nécessite qu'une indirection effective. Elle donne l'adresse de mapping du segment à laquelle est ajoutée un déplacement fonction des numéros de page et d'objet. La taille de l'identifiant est un paramètre d'instanciation du système: il peut être codé sur 32 bits ou sur 64 bits suivant la quantité d'objets créés durant la vie de cette base. Le codage sur 32 bits offre une base modulable entre 32 Go d'objets courts ou 4 To d'objets longs.

Interface Langage: le typage des objets.

Le rôle de l'interface Langage est de répertorier dans la métabase les classes C++ utilisées par les WorkSpaces et d'instancier ces classes dans les objets courts. Cette interface utilise un parser, un compilateur C++ du marché (ATT ou GNU) et les outils de développement standards associés (browser, debugger). Le parser ajoute les nouvelles classes persistantes à la métabase et génère le code de quelques méthodes réalisant notamment la traduction des identifiants d'objet.

Contrôle de Concurrence: le Callback Locking hiérarchique.

Le **Callback Locking** (CL) évalué par [Wang91], est une méthode de verrouillage 2 phases qui maintient sur le client un cache des verrous en lecture obtenus par les transactions. La demande du verrou en écriture nécessite de relâcher le verrou en lecture dans le cache de chaque client. Le WorkSpace serveur centralise et propage demandes et relâches de verrous. Cette méthode est particulièrement adaptée aux WorkSpaces dans lesquels les transactions d'un même client référencent les mêmes objets. Comme le WS serveur ne connaît pas l'état des verrouillages dans les clients, les interblocages possibles sont résolus par time-out [DeWi90].

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 distribués sur un réseau de stations 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 Octobre 1993 sur Solaris 2.2. 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

[ATT93] ATT Bell Laboratory, "ODE 2.0 User's Manual", 1993

[Deux90] O. Deux et al, "The Story of O2", IEEE Transactions on Knowledge and Data Engineering, Vol 2, No 1, March 1990.

[DeWi90] D. J. DeWitt et al, "The Gamma Database Machine Project",IEEE Transactions on Knowledge and Data Engineering, Vol 2, No 1, March 1990.

[Garz88] J F.Garza, W Kim, "Transaction Management in Object-Oriented Database System", SIGMOD 1988.

[IEEE92] IEEE, "POSIX 1003.4a Draft 6 - Threads Extension for Portable Operating Systems", Feb 1992.

[Khos86] S. Khoshafian and G. Copeland, "Object Identity", MCC Research Report, 1986.

[Lamb91] Charles Lamb, Gordon Landis, Jack Orenstein, Don Weinreb, "The ObjectStore Database System", Communication of the ACM October 1991, Vol. 34, No. 10

[Sun93a] SunSoft, SunOS 5.2 Guide to MultiThread Programming, Answer Book, March 1993

[Sun93b] SunSoft, SunOS 5.2 Memory Management, Answer Book, March 1993.

[Wang91] Y. Wang, L. A. Rowe, "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture", SIGMOD 1991