Chapitre 2

Etat de l'art

I. Introduction

Le domaine des Gérants d'Objets Persistants est issu de la convergence des recherches menées dans les domaines des Bases de Données et des Langages Orientés-Objets. Les Gérants d'Objets Persistants sont plus particulièrement adaptés dans la manipulation des volumes importants de données complexes : ils autorisent une plus grande productivité en soulageant le développeur d'applications de gestion de la persistance ou de la concurrence des accès, mais ils lui offrent aussi un modèle complexe de données et un langage général de manipulation de celles-ci. Ce modèle et ce langage ne l'obligent pas à faire des compromis comme dans le cas des SGBDs relationnels.

Dans cet état de l'art, nous présenterons plusieurs Gérants d'Objets qui sont des produits commerciaux ou des prototypes de recherche. Nous ne ferons pas une description exhaustive de chacun de ces systèmes mais nous insisterons sur les points qui particularisent dans la gestion des objets dans ceux-ci. Les systèmes, tels que ODE ou O2, proposent des fonctionnalités intéressantes comme des bases actives [Biliris93] ou des interfaces multiples [Lécluse88, Deux90]. Cependant l'étude des aspects fonctionnels des gérants d'objets représenterait un autre état de l'art qui ne peut tenir ici pour des raisons de place. Le lecteur pourra se référer aux articles suivants [Popovitch91, Ahmed92, Amiel92] pour avoir un panorama des fonctionnalités étendues de ces systèmes.

Dans les trois premiers systèmes, nous nous intéresserons à l'architecture de ces gérants d'objets qui proposent d'échanger des pages ou des objets entre les clients et le serveur. Avec Texas et QuickStore, nous découvrirons deux techniques de "swizzling" qui convertissent les références persistantes. Cricket et Eos proposent d'accès aux données persistantes au travers d'une mémoire virtuelle transactionnelle. Ensuite, nous ne verrons d'ObjectStore que son mécanisme de version de la base. Avec Shore, nous verrons une architecture nouvelle de Gérant d'Objets et les mécanismes de contrôle de concurrence adaptatif. Enfin, nous terminerons par l'architecture applicative du Newton, le PDA d'Apple, dont la structure est très similaire aux Espaces de Travail dont cette thèse fait l'objet.

II. Exodus

Exodus a été développé à l'Université du Wisconsin-Madison (avec Cricket présenté en section VI et SHORE présenté en section VIII). Exodus est un des premiers gérants d'objets extensibles et comporte des développements récents à travers QuickStore (section V) qui utilise les mécanismes des systèmes d'exploitations comme d'autres approches (ObjectStore). Par contre, il présente des faiblesses sur le plan des fonctionnalités et des interfaces. Il sert de système étalon dans les comparaisons du benchmark OO7 [Carey93].

Exodus est constitué des deux composants principaux : le système de stockage ESM et l'interface langage E. Le système de stockage est basé sur l'architecture de serveur de pages dans laquelle les processus clients demandent les pages au serveur. Si la page n'est pas présente dans le buffer du serveur, celui-ci effectue une lecture disque puis retourne la page demandée au client. Le système de stockage s'occupe également du contrôle de concurrence et de la reprise sur panne [Franklin92]. Le technique de contrôle de concurrence est le verrouillage 2-phase

(2PL) avec deux granularités possibles : la page ou le fichier. Les pages d'index utilisent un protocole spécial différent du 2PL. Les pages sont cachées par le client d'une transaction à l'autre, cependant le verrou doit être réclamé par chaque nouvelle transaction client au serveur.

L'interface Langage E est définie sur le modèle du C++. La persistance est ajoutée par un pré-compilateur (basé sur le compilateur GNU g++) sous la forme d'appel à des fonctions à une machine virtuelle, l'EPVM [Schuh90]. L'EPVM gère un buffer de pages dans la mémoire du processus client; l'accès aux objets dans les pages de ce buffer passe donc par des fonctions qui maintiennent la page en mémoire durant la durée de l'accès; la page peut être ensuite chassée du buffer si de la place est nécessaire. L'appel à ces fonctions lors de chaque accès à un objet se révèle être très coûteux.

La dernière version de l'EPVM permet de maintenir dans le buffer un certain nombre de pages résidentes en mémoire; il autorise ainsi l'utilisation du *swizzling at dereferenciation* pour accélérer la traversée des références persistantes [White92] : Quand une référence est traversée pour accéder aux champs d'un objet, le swizzling convertit celle-ci (qui est contenue dans un objet) vers l'adresse mémoire de l'objet référencé; les traversées suivantes utilisent l'adresse directe de l'objet. Cette méthode a cependant l'inconvénient de nécessiter d'effectuer la conversion inverse (*l'unswizzling*) pour toutes les références swizzlées du buffer quand une des pages résidentes en mémoire en chasser de celle-ci. L'unswizzling est effectué dans les pages modifiées renvoyées au serveur au moment de la validation de la transaction.

III. Versant

Versant utilise une architecture client-serveur orienté vers l'échange et le verrouillage d'objets. Le serveur n'utilise la notion de page que pour stocker les objets dans les disques; le reste du système ne manipule que des objets individuels. Cette approche offre un grain fin de manipulation des données : le verrouillage au niveau des objets évite des contentions d'accès inutiles entre les transactions que l'on trouve dans les systèmes à serveurs de pages. Cependant le chargement objet par objet sur le client entraîne un surcoût dans les communications dans les réseaux locaux. En effet dans un tel contexte, transférer une page représente sensiblement le même coût que de transférer un objet ou un verrou. Si une transaction accède à des N objets regroupés dans la même page, le serveur d'objets procède à N échanges avec le client au lieu d'un seul avec un serveur de pages, avec un coût sensiblement N supérieur. Versant partage ce principe d'architecture serveur d'objets avec d'autres systèmes comme Orion/Ithasca [Kim90] et Gemstone [Butterworth91].

IV. Ontos

Ontos [Ontologic90] a une approche unique qui utilise de manière combinée les deux types d'architecture de serveur; une application peut créer un objet sur un serveur d'objet ou bien le créer groupé avec d'autres objets dans une page sur un serveur de page. Ces deux types d'instanciation peuvent coexister dans la même application. Le grain de concurrence est l'objet quand celui-ci provient du serveur d'objets et la page pour un objet groupé avec d'autre. Le

processus client bufferise les objets (et les pages) dans son espace d'adressage. Cette approche limite la quantité d'objets accessible au cours d'une transaction à la taille du swap de la station de travail. Les objets (ou pages) modifiés sont d'abord tous journalisés avant d'être recopiées sur l'archive; cette approche simplifie l'opération de reprise sur panne qui s'effectue en ré-appliquant les dernières modifications journalisés (i.e. REDO).

Ontos possède une autre originalité : il propose à la fois une méthode optimiste et une méthode pessimiste pour le contrôle de la concurrence. Ontos permet aussi d'effectuer des checkpoints écrivant les modifications sur l'archive en cours de transaction; cette approche peut être utilisée pour implanter un verrouillage du type "un seul écrivain et plusieurs lecteurs".

V. Texas & QuickStore

Ces deux gérants d'objets sont des prototypes de recherche. Texas (Université d'Austin) est actuellement un système mono-utilisateur alors que QuickStore utilise l'architecture Client-Serveur d'EPVM (Exodus). Ces deux systèmes utilisent des techniques de swizzling qui permet de manipuler les objets persistants sans modifier le code des applications. Contrairement à la technique de swizzling "at déréférenciation" que nous avons déjà vu dans Exodus, la technique de swizzling utilisée consiste à convertir toutes les références persistances contenues dans les objets d'une page au moment du chargement de la page dans l'espace d'adressage du processus. Cette technique utilise l'adressage virtuel et la protection des zones mémoires proposés par le noyau du système (Memory Management Unit).

Dans Texas [Singhal92 ,Wilson92], la transaction commence par référencer un objet "racine" du graphe d'objets. Cette référence provoque une violation de protection de la mémoire virtuelle; la page est alors chargée depuis le disque dans cette zone mémoire et les références persistantes contenues dans la page sont converties vers les adresses virtuelles des pages contenant objets référencés. Si une (ou plusieurs) de ces pages n'a pas encore été accédée, un emplacement, protégé contre tous les accès, lui est réservé dans l'espace d'adressage; une référence dans cet emplacement provoque le chargement et le swizzling de cette page. Lors de la validation de la transaction, les pages modifiées doivent être "unswizzlées" avant d'être enregistrée sur l'archive. L'unswizzling consiste à convertir les références swizzlées vers les identifiant des objets référencés; ceci est nécessaire car les adresses virtuelles ne sont valides que durant les processus.

Pour éviter le coût du swizzling au chargement et de l'unswizzling lors d'une validation, QuickStore [White94] qui s'inspire d'ObjectStore sur ce point, propose d'enregistrer les pages d'objets avec leurs références swizzlées. Le chargement d'une page dans l'espace d'adressage nécessite d'installer celle-ci à l'adresse à laquelle elle était installée lors de la session précédente. Néanmoins, ceci n'est pas toujours possible; la page installe en mémoire doit être re-swizzlée si une des pages référencées a été chargée à un autre emplacement. Cette stratégie nécessite de conserver d'une session à l'autre pour chaque page les adresses virtuelles d'installation des pages qu'elle référence; cette information est stockée dans des objets persistants, les "mapping objects", rangées dans des pages indépendantes de la base (ces "meta-objets" profitent ainsi du service de reprise sur panne). Cette stratégie compte beaucoup sur la possibilité de replacer les pages aux mêmes emplacements du buffer de pages (et dans l'ensemble de la mémoire virtuelle pour ObjectStore).

Nous terminons cette section par une remarque sur l'organisation disque de Texas. Le "Log Structure Storage" (LSS) est fondée sur la constatation que le trafic vers les disques est principalement dominé par des écritures synchronisant les buffers mémoires avec les disques. LSS propose donc de réaliser les écritures disques séquentiellement afin d'améliorer le débit d'écriture disque. Avec cette démarche, une page modifiée n'est pas réécrite dans le même bloc physique que l'image précédente; il n'est donc pas nécessaire d'utiliser le protocole Write Ahead Log (majoritairement utilisé par les Gérants d'Objets) qui impose de séquencer la journalisation et l'écriture sur l'archive. L'ancienne image de la page est réclamée une fois que la validation de la transaction est terminée. LSS s'inspire largement de l'organisation du système de fichier LFS de Sprite [Rosenblum92] qui n'utilise pas la notion de transaction pour réclamer les anciennes images. Cette organisation ne tient cependant pas compte des longues lectures séquentielles fréquentes pour des données Multimédia.

VI. Cricket & Eos

Cricket est un des prototypes de recherche de l'Université de Wisconsin-Madison [Shekita90a,b] et Eos est en cours de développement à l'INRIA [Gruber92a,b]. Ces deux systèmes se placent dans la perspective des "grands" espaces d'adressage offerts par la nouvelle génération de processeurs [Chase92, Seznec92]. Dans leur approche commune, les identifiants d'objets sont tous simplement des adresses de la mémoire virtuelle.

L'implémentation de ces systèmes est réalisée au-dessus du micro-noyau Mach [Baron87] et utilise une fonctionnalité particulière de ce système pour accéder aux données persistantes [Young87, Rashid87]. Mach autorise les processus clients à mapper dans leur espace d'adressage un "memory object". Un processus externe appelé External Pager est chargé de contrôle les mouvements entre la mémoire physique alloué au processus et le "memory-object" qui peut être un fichier. Dans l'approche bases de données de Cricket et Eos, le "memory-object" est la base et l'External Pager remplit le rôle de serveur de page et de verrou.

VII. ObjectStore

ObjectStore est le système commercial d'ObjectDesign Inc. [Lamb91, ODI93]. ObjectStore est architecturé sur le modèle Client/Serveur. Le serveur stocke et retrouve les pages de données sans connaître leurs contenus, suivant les requêtes des clients. Le serveur est responsable des services de reprise sur panne et de contrôle de concurrence. Le verrouillage par rappel (Callback Locking) autorise les clients à cacher les verrous (en consultation) d'une transaction à l'autre évitant les demandes répétées au serveur. Une demande de verrou exclusif nécessite l'invalidation des pages répliquées dans les caches des clients. Pour accéder aux objets, la transaction utilise des pointeurs dans la mémoire virtuelle du processus client; ObjectStore utilise une méthode de swizzling semblable à celle de QuickStore.

ObjectStore est un système riche en fonctionnalités; toutefois nous nous intéresserons dans la suite de cette section qu'aux mécanismes de versions mis en oeuvre par ObjectStore. Ce mécanisme de version offre des facilités pour implanter le travail de groupe; un utilisateur peut

charger (check-out) dans son espace de travail privé (Workspace)[1] une version alternative d'un objet (ou d'un groupe d'objets) et consulter ou modifier cette version; cet utilisateur n'est donc pas gêné par la concurrence d'un autre utilisateur qui consulte et modifie simultanément une autre version de l'objet dans son espace de travail. Une fois le travail terminé, les objets modifiés des versions sont déchargés (check-in); les versions alternatives produites peuvent être ultérieurement fusionnées ensemble pour "réconcilier" les différences. Cependant, la fusion, qui est une opération spécifique à une application, doit être écrite par le développeur.

ObjectStore permet de créer un espace de travail en héritant les modifications d'un autre; ce dernier peut ainsi être partagé par plusieurs espaces de travail privés. Cette possibilité d'imbrication permet de former une hiérarchie de groupes et de sous-groupes de travail dans lesquelles plusieurs utilisateurs peuvent coopérer au moyen d'un espace privé et d'un check-in rendant les modifications publiques dans l'espace partagé qui représente le groupe.

VIII. Shore

Le système SHORE (Scalable Heterogenous Object REpository) de l'Université de Wisconsin-Madison [Carey94] propose une approche nouvelle dans le domaine des gérants d'objets. Sa conception a démarré à partir des constatations suivantes qui semblent freiner l'utilisation des gérants d'objets comme système de stockage :

- Ils sont souvent trop fortement couplés avec leur langage d'interface et ne proposent pas d'autres inerfaces.
- Ils ne permettent pas de réutiliser les applications initialement prévues pour archiver leurs données sur des fichiers.
- Ils proposent rarement d'autres modèles d'exécution que celui des transactions sérialisables.
- Ils sont mal adaptés à la construction de serveurs applicatifs.

Le modèle de données de SHORE considère des objets de taille moyenne, c'est à dire qu'ils sont plus petits que des fichiers UNIX mais qu'ils sont trop gros pour constituer les objets noeuds de structures persistantes comme des listes ou des arbres. Les objets sont identifiés par un OID unique. La structure de l'objet contient des valeurs simples ou structurées qui peuvent inclure des références OID vers d'autres objets. Chaque objet référence un objet type définissant l'interface. La structure de l'objet comprend une partie fixe (core) et optionnellement une partie dynamique (heap) qui se comporte comme un tas persistent. Ce tas contient des sous-objets permettant de définir des structures telles que des listes, des arbres un peu à la manière d'O2. Les références internes au tas sont des déplacements par rapport à la base de ce tas quand les sous-objets sont sur le disque, mais elles sont swizzlées une fois en mémoire. Chaque tas d'objets se comporte comme une base d'objets. SHORE propose un nommage externe de ces objets semblables aux hiérarchies de fichiers des Systèmes de Gestion de Fichiers (SGF) traditionnels. L'accès aux objets peut donc se faire aussi grâce à l'interface des SGFs (open,

---

1 La notion d'Espace de Travail présentée dans ce thèse dépasse la notion de Workspace proposée par ObjectStore. ObjectStore se limite à définir un espace logique pour modifier une version alternative.

read, write, close); ces opérations ne consultent que la partie dynamique des objets nommés en considérant que celle-ci est une chaîne sans structure à la manière d'un fichier UNIX. Cette compatibilité avec les SGFs permet de réutiliser les applications écrites pour ceux-ci dans le Gérant d'Objets.

L'architecture de SHORE est basée sur une structure *symétrique pair-vers-pair*. Chaque machine de réseau est équipée d'un serveur SHORE. Une application cliente contacte son serveur local pour obtenir un objet. Celui-ci recherche l'objet dans son cache de page; si la page contenant l'objet ne se trouve pas dans le cache local, la page est soit chargée depuis le disque, soit demandée au serveur SHORE distant qui l'archive. Le serveur SHORE se divise en deux composants principaux : l'Interface Server qui dialogue avec les applications et le Storage Manager qui gère les objets persistants. L'Interface Server est en fait une couche de service qui exporte vers les applications clients les objets qu'il accède par le biais du Storage Manager. D'autres couches de services peuvent être développées et côtoyer l'Interface Server sur un même serveur local pour l'accès aux objets; ces couches de services prennent le nom de Service à Valeur Ajoutée (VAS: Value-Added Server). SHORE propose de cette manière un service VAS de serveur de fichiers (i.e. sur la partie variable des objets nommés) suivant le protocole NFS. Nous verrons dans le chapitre suivant que cette architecture permet de résoudre le problème du serveur à valeur ajoutée implanté par un client du serveur d'objets (ou de pages).

Coté interface, SHORE propose un langage de définition des structures (SDL) indépendant de tout langage de programmation courant; Sa syntaxte est celle de l'interface recommandé par l'ODMG. Il propose un large éventail des types simples, de constructeur de type ainsi que des types association déjà présent dans ObjectStore, etc. Le SDL peut être alors traduit vers n'importe quel langage et notamment le C++. Dans l'interface C++, les verroux exclusifs doivent être posés explicitement par le développeur.

SHORE utilise une méthode de contrôle de concurrence originale qui adapte le grain d'un verrou en fonction du degré de contention sur celui-ci [Carey94b]. Cette méthode est basée sur le verrouillage 2 phase avec réplication des pages dans les caches des serveurs (i.e. le Verrouillage par Rappel ou Callback Locking). Le verrou sur un objet est demandé au serveur de l'objet; le serveur accorde soit le verrou au client demandeur (qui est un autre serveur Shore) si le verrou demandé est compatible avec le verrou courant de l'objet. Pour éviter les demandes successives, le serveur envoie en même temps les verrous des autres objets de la page (excepté ceux verrouillés en écriture). Le rappel de verrous (c'est à dire, l'invalidation des réplicas dans les caches) utilise également un grain adaptatif : le serveur demande l'invalidation d'un objet de la page; le client a le choix d'invalider l'objet si des verrous en lecture existent pour d'autres objets de la page ou bien d'invalider la page complètement si aucun verrou n'est posé dans la page.

La validation suit le protocole 2-phase pour une transaction uniquement si les objets accédés étaient maintenues par plusieurs serveurs. Le serveur local coordonne ce commit. Le recovery utilise les techniques d'ARIES. Au commit, le client retourne au serveur local les enregistrements de journal des objets modifiés qui les redirige vers le serveur possesseur. Ce dernier re-applique (redo-at-server) les modifications sur ces objets locaux. Le client ne peut alors jamais modifier les méta-structures. Cette technique fait perdre l'avantage du data-shipping qui décharge le serveur et sera comparée avec celle classique d'EXODUS.

IX. Newton d'Apple

Nous trouvons intéressant d'ajouter à cette liste le Newton MessagePad d'Apple. En effet, l'architecture applicative de ce PDA (Personnal Digital Assistant) ressemble beaucoup à la structure du Workspace que nous étudions dans cette thèse. Le Newton est le premier PDAs qui offre une plate-forme de développement qui permettra de l'utiliser dans le cadre des Bases de Données Nomades.

Le système d'exploitation du Newton [Welland94] est conçu sur le modèle d'un seul espace d'adressage partagé les multiples activités du système; néanmoins le système mais en place la notion de domaine qui regroupent les données propres à chaque activité (pile et données privées). Les applications s'exécutent donc dans le même espace d'adressage; elles font donc partie d'une "méta-application" et peuvent ainsi partager le code (organisé en package) et les données par simple échange de pointeur.

Le Newton utilise une gestion transparente d'objets persistants [Smith94]; les objets sont chargés dans l'espace d'adressage par une simple référence sur ceux-ci. Le système de stockage est la mémoire primaire (sauvegardée par une pile lithium) et des cartes additionnelles de mémoire flash qui archivent les objets regroupés par ensemble appelé "soup"). Un objet peut être archivé dans un format compressé; il sera décompressé avant d'être installé en mémoire. Bien que le Newton utilise une MMU raffinée pour la gestion des domaines dans l'espace d'adressage, celle-ci n'est pas utilisée pour réaliser des défauts d'objets entre la mémoire et la zone de stockage : cette opération est faite au moyen d'handler (les "fault blocks"). La gestion de la persistance est faite suivant le modèle transactionnel; les données modifiées au cours d'une transaction sont conservées en mémoire primaire jusqu'au moment de la validation ou de l'abandon de la transaction.

X. Vroom

Le projet VROOM mené dans l'équipe RAPID du laboratoire MASI, nous a permis d'expérimenter différentes architectures et de constater les limitations du modèle Client-Serveur. VROOM a fait l'objet de plusieurs implémentations dont les plus récentes ont été utilisées dans des projets industriels.

VROOM V0 est une version libraire mono-utilisateur qui propose au développeur une API pour réaliser la persistance sur des données locales. Les versions V1 de VROOM sont des architectures basées Client-Serveur dont le grain de communication est la page. Ces versions offrent une interface C++ transparente dans laquelle le développeur n'explicite pas la gestion de la persistance et du verrouillage. Cette interface est implantée par un pré-compilateur compatible avec n'importe quel compilateur C++ du marché. Le client met en oeuvre un double buffer de pages et d'objets comme dans certaines versions d'ORION [Kim90]. Le serveur gère un buffer de page ainsi que le contrôle de concurrence et la journalisation. Le contrôle de concurrence propose le verrouillage 2-phase classique ainsi qu'un assouplissement qui autorise des lectures sales ("un écrivain et plusieurs lecteurs").

VROOM V1 a été utilisé pour des applications de Génie Logiciel dans le projet Eureka/Genelex de gestion de dictionnaire et dans le projet Esprit F3 ("From Fuzzy to Formal").