Gestion des informations persistantes dans un système réparti à objets

Rodrigo Scioville Garcia

THESE

présentée par

Rodrigo SCIOVILLE GARCIA

pour obtenir le titre de **DOCTEUR**

de L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

(arrêté ministériel du 5 juillet 1984)

Spécialité : INFORMATIQUE

GESTION DES INFORMATIONS PERSISTANTES DANS UN SYSTEME REPARTI A OBJETS

Thèse soutenue le 23 juin 1989

Composition du Jury :

Président :	M. Jacques MOSSIERE
Rapporteurs :	M. Michel ADIBA
	M. Marc SHAPIRO
Examinateurs :	M. Roland BALTER
	M. Sacha KRAKOWIAK

Thèse préparée au sein du Laboratoire de Génie Informatique
à l'Université Joseph Fourier - Grenoble 1

Je tiens à remercier :

Monsieur Jacques Mossière, Directeur de l'Ecole Nationale Supérieure d'Informatique et Mathématiques Appliquées de Grenoble, de m'avoir fait l'honneur de présider le jury de cette thèse.

Monsieur Michel Adiba, Professeur à l'Université Joseph Fourier, pour les conseils qu'il m'a donnés lors de la rédaction de ce document et pour avoir accepté de participer au jury.

Monsieur Marc Shapiro, Directeur de Recherche à l'Institut National de Recherche en Informatique et Automatique, d'avoir bien voulu rapporter ce travail et de faire partie du jury.

Monsieur Roland Balter, Directeur du Centre de Recherche Bull à Grenoble, pour l'intérêt qu'il a porté à la réalisation de ce travail et pour sa participation au jury. Je tiens à lui témoigner toute ma reconnaissance pour le soutien permanenet qu'il m'a manifesté.

Monsieur Sacha Krakowiak, Professeur à l'Université Joseph Fourier, pour m'avoir accepté dans son équipe et constamment soutenu dans mes recherches.

Monsieur Gérard Vandôme, Ingénieur au Centre de Recherche Bull, pour les conseils qu'il m'a prodigués lors de la réalisation de ce travail. Sans les discussions que nous avons eues, cette thèse n'aurait sans doute jamais vu le jour.

Messieurs André Freyssinet et François Exertier, Thésards au Centre de Recherche Bull, pour l'aide inappréciable qu'ils m'ont apportée durant l'élaboration de cette thèse.

Monsieur Philippe Chol pour la patience de relire et de corriger la dernière version de ce document.

Mes collègues des projets Guide et Comandos, du Centre de Recherche Bull et du Laboratoire de Génie Informatique, pour leur aide et leur sympathie. Je voudrais remercier particulièrement Messieurs Christian Lenne et Jacques Bernadat du Centre de Recherche Bull et Monsieur Bernard Cassagne du Laboratoire de Génie Informatique.

Le Centre Régional des Oeuvres Universitaires de Grenoble, l'Ambassade de France en Colombie et le Departamento de Sistemas y Computación de la Universidad de los Andes de Bogotá, qui ont rendu possible ma venue et mon séjour en France.

Je tiens à exprimer toute ma reconnaissance et ma gratitude à :

Monsieur Eric Paire, Ingénieur au Centre de Recherche Bull, non seulement pour son support technique mais surtout pour la compagnie et le soutien constants qu'il m'a donnés lorsque nous partagions le même bureau. Sa présence m'a permis de supporter mieux les longues soirées d'hiver consacrées à la rédaction de cette thèse.

Madame Marie Meysembourg-Männlein, Thésard au Laboratoire de Génie Informatique, pour avoir lu et commenté tous mes écrits, pour les agréables discussions que nous avons eues, et surtout pour ses encouragements, ses conseils et sa grande sympathie.

Mes copains Latinoaméricains et Européens, pour la patience, la solidarité et l'amitié qu'ils m'ont toujours témoignées.

Résumé

Ce travail aborde quelques-uns des problèmes de la gestion d'objets dans un système informatique réparti.

Guide est un système d'exploitation expérimental réparti sur un réseau local. Les informations qu'il manipule sont structurées selon un modèle à objets. Le concepteur d'applications voit le noyau de ce système comme une machine virtuelle multi-sites et multi-processeurs dans laquelle le parallélisme est apparent et la distribution est cachée. Cette machine virtuelle réalise la gestion d'objets.

Ce travail décrit l'un des composants de la machine virtuelle : la mémoire d'objets. Il présente les choix de conception, sa réalisation et une analyse de l'expérience acquise. La mémoire d'objets est persistante. En effet, les objets ne sont pas détruits tant qu'ils restent liés à un objet particulier, persistant par définition. La mémoire d'objets est répartie et comporte deux niveaux : la Mémoire Virtuelle d'Objets, qui constitue le support d'exécution des objets, et la Mémoire Permanente d'Objets, qui se charge de leur conservation.

Certaines fonctions de gestion d'objets n'ont pas été intégrées dans la machine virtuelle mais elles sont mises en oeuvre comme des applications. On appelle ces applications les services du système. La conception et la réalisation des services de désignation symbolique et de gestion de versions sont décrites dans cette thèse.

Table des Matières

Introduction

1. Description du cadre de travail

Dans un système informatique, on dit qu'une donnée est **persistante** lorsque sa durée de vie dépasse celle de la procédure ou du processus qui l'a créé. Autrement, la donnée est **temporaire**. La conservation d'une donnée persistante est assurée en lui fournissant une représentation en mémoire secondaire.

D'autres processus peuvent, par la suite, utiliser et éventuellement modifier cette donnée persistante. Toute modification doit être intégrée dans sa représentation en mémoire secondaire. Ces actions ont lieu à plusieurs reprises jusqu'à la destruction de la donnée.

Persistance 'classique'

Dans les langages à blocs, comme Pascal et C, la durée de vie du contenu d'une variable ne dépasse pas celle d'un bloc d'exécution, à moins que la variable soit recopiée dans un fichier. Il faut pour ceci utiliser explicitement une commande d'écriture. De manière similaire, on ne peut accéder au contenu d'un fichier que par l'exécution explicite d'une commande de lecture. Il existe des langages dans lesquels ces deux opérations ne peuvent se réaliser que sur certains types de variables. C'est le cas du type `record` associé au type `file` de Pascal.

La persistance dont certains langages à tas, comme Lisp et Prolog, dotent leurs variables est mise en oeuvre d'une manière assez différente. Lors de l'exécution d'un programme écrit dans l'un de ces langages, le tas est rangé dans un fichier en fin de session. Cette représentation du tas est rendue disponible pour exécution lors des nouvelles sessions. A cause des adresses qu'il contient, le tas doit être rechargé toujours au même emplacement de mémoire.

Dans la mesure où l'une de ses principales fonctions est la conservation des données, tout système de gestion de bases de données (SGBD) fournit la persistance de manière implicite. Cependant, dans les programmes écrits en langages hôtes, une différence claire et nette existe entre les variables 'normales' et celles dont le contenu provient d'une base de données.

Nous appelons persistance 'classique' celle que fournissent les sytèmes de gestion de fichiers et les SGBDs dont les données sont structurées selon l'un des trois modèles traditionnels. Les données temporaires et persistantes sont traitées de manière différente ; en particulier, la destruction d'une donnée persistante se réalise explicitement.

Des efforts ont été entrepris dans les dernières années pour atténuer, voire éliminer, les contraintes qu'impose la persistance classique. Une première approche consiste à

déclarer persistantes les variables manipulées par le biais d'un langage de programmation. La conservation du contenu de ces variables est implicite : les lectures et écritures en mémoire secondaire se font sans intervention de l'utilisateur. Cette voie a été suivie par Butler pour rendre persistantes les variables de ses programmes Lisp [Butler86].

Une deuxième ligne de recherche est celle de la 'programmation persistante'. L'idée de base consiste à définir la persistance des données par rapport à une donnée spéciale, qui est par définition persistante. Ainsi, une donnée est persistante tant qu'il existe un lien entre elle et la donnée spéciale.

Avant de rentrer dans les détails de la programmation persistante, nous voulons nous arrêter brièvement sur les systèmes à objets, notamment sur les systèmes de programmation à objets. En effet, la programmation persistante a été conçue et développée pour des systèmes de programmation à objets. En particulier, ce sont les pointeurs entre objets qui permettent la mise en oeuvre de la notion de 'lien entre données'.

Les systèmes à objets

Le déroulement d'un système à objets, de même que celui de n'importe quel système, peut être vu comme une séquence d'états. Un état comprend l'ensemble des valeurs des propriétés des composants du système à un moment déterminé. Chaque état est obtenu par la transformation d'un état antérieur. D'après Nygaard [Nygaard86], le trait particulier des systèmes à objets est que les transformations correspondent aux actions que chaque composant, appelé 'objet', provoque sur ses propriétés individuelles et sur celles des autres objets.

Un langage à objets, de même que n'importe quel langage de programmation, comporte un mécanisme d'abstraction. Toujours selon Nygaard, ce mécanisme fait d'un type l'abstraction d'un ensemble de propriétés et d'une classe l'abstraction du comportement d'un ensemble d'objets à la suite des receptions de messages[1], qui sont l'abstraction des actions. On peut remarquer que, dans un objet, un lien (pointeur ou référence) est une propriété dont la valeur est un autre objet.

En contraste avec la présentation de Nygaard, qui parle des propriétés internes, Rentsch propose une caractérisation externe des objets qu'un langage de programmation à objets permet de manipuler [Rentsch82]. En s'inspirant des concepts de Smalltalk™, il affirme que tout objet possède les propriétés externes suivantes :

---

1. Nygaard utilise le terme 'appel de procédure' à la place de celui d''envoi de message', qui a été consacré par Smalltalk.

™ Smalltalk est une marque déposée de *Xerox Corporation.*

- Uniformité d'apparence et de statut. Tous les objets sont égaux. La différence entre les objets 'de l'utilisateur' et ceux 'du système' n'existe pas.
- Uniformité de référence. Tout objet possède un identificateur qui permet de le désigner de manière unique. Un identificateur quelconque peut désigner un objet quelconque car les identificateurs sont tous du même genre.
- Uniformité dans le mode de communication. Lorsqu'un objet veut qu'un autre objet réalise une action, il lui envoie un message.
- Capacité de réaliser une action. Tout objet est capable de réaliser l'ensemble d'actions qui définit son comportement. A la reception d'un message, l'objet réalise l'action demandée et rend un résultat.

Les autres propiétés, celles dont Nygaard parle, définissent la structure des objets et les manières d'accéder à cette structure. Si la dernière propriété externe indique qu'un objet peut réaliser une action, ce sont les propriétés internes qui disent comment le faire. Les propriétés internes d'un objet sont invisibles aux autres objets : le concept d'ouvrir un objet, comme l'on fait avec un fichier, n'existe pas.

Une dernière propriété externe des objets est le partage. Tout objet à la possibilité de partager avec d'autres certaines propriétés internes afin de se comporter de la même manière. En contrepartie, tout objet est libre d'adapter à ses propres besoins certaines des propriétés partagées afin d'avoir un comportement individuel. La mise en oeuvre de cette propriété constitue le mécanisme d'héritage.

Comme conséquence de son grand pouvoir d'abstraction, le paradigme des objets est devenu aujourd'hui l'instrument qui permet à des disciplines aussi diverses que les langages de programmation, les bases de données, les systèmes d'exploitation et les systèmes experts non seulement d'exprimer leurs besoins individuels mais aussi de manifester leurs points communs. Il constitue donc le point de départ pour la conception et le développement d'un éventail de systèmes dont le nombre ne cesse de s'accroître.

L'utilisation de la programmation à objets s'est largement répandue comme résultat du développement du langage Smalltalk [Goldberg76] [Ingalls78] [Goldberg83] [Deutsch84]. Bien que sa structure interne, surtout en ce qui concerne les pointeurs entre objets, ait été largement influencée par Lisp, le concept de 'classe', emprunté à Simula, domine la conception de Smalltalk. Une analyse approfondie des caractéristiques des langages à objets se trouve dans [Meysembourg89].

Le paradigme des objets a été également utilisé dans les recherches sur les systèmes d'exploitation. Les systèmes Eden [Almes85] [Black85], Emerald [Black86a] [Black86b] et Argus [Liskov83] [Liskov88] ont montré la puissance d'un modèle à objets pour la solution des problèmes posés par la répartition. L'objectif du système SOS [Shapiro85] est la mise en place d'un environnement bureautique réparti. Par l'utilisation d'un modèle à objets, la structure de cet environnement est modulaire et modifiable dynamiquement.

Deux tendances se dessinent dans les SGBDs qui commencent à paraître aujourd'hui : Les SBGDs extensibles et les SGBDs à objets. Les premiers tirent partie d'un modèle à objets pour assurer plus facilement l'extensibilité. Leur principe, illustré par les systèmes Exodus [Carey86] et Geode [Pucheral88], est celui d'une boîte à outils qui permet de construire des SGBDs adaptés à une ou plusieurs applications. Les SGBDs à objets sont ceux dont le modèle de données est un modèle à objets. Pour eux, l'extensibilité est l'un des avantages fournis par leur modèle. Des exemples sont les SGBDs O_2 [Bancilhon88] et Iris [Fishman87].

Quels que soient ses objectifs particuliers, tout système à objets réalise deux fonctions bien précises : la mise en oeuvre d'un modèle à objets et la gestion d'objets. La première comprend la mise en place des types et classes ainsi que la création d'objets en tant qu'instances de ces classes. Elle concerne aussi la réalisation de l'héritage et des règles selon lesquelles un objet d'un type peut être remplacé par un objet d'un autre (conformité de types).

La gestion d'objets prend en compte l'exécution des actions d'un objet à la suite de la réception des messages, tout en contrôlant l'accès à l'objet et sa cohérence. Elle se charge en outre de la création, la désignation, la localisation et la conservation des objets du système. Par analogie avec les systèmes de gestion de fichiers, dont les fonctions rappellent en maints aspects quelques-unes de la gestion d'objets, l'ensemble de ces fonctions est souvent appelé le 'système de gestion d'objets' (SGO ou gestionnaire d'objets).

Programmation persistante

Les premiers systèmes à objets se sont distingués par la dichotomie existante entre les deux niveaux de mémoire. En effet, tandis que la mémoire virtuelle a été rapidement pourvue de fonctions de gestion de mémoire assez développées, comme le ramassage automatique des miettes, la mémoire secondaire a été limitée, en général, à son rôle de conservation. S'il existe des systèmes qui ont enrichi leur mémoire secondaire de fonctions telles que le support à la consultation rapide de données et le maintien de la cohérence, il y en a d'autres qui n'ont même pas intégré la fonction de conservation. C'est le cas de Smalltalk-80 standard, où tous les objets du système se trouvent en mémoire principale [Goldberg83].

Certains systèmes à objets ont cependant ressenti le besoin de la persistance. La gestion d'objets se chargeant désormais de l'interface avec la mémoire secondaire, les utilisateurs disposent d'un nouveau moyen pour indiquer la persistance de leurs objets : la programmation persistante. Programmer de 'manière persistante' consiste à utiliser un ensemble d'outils qui permettent de lier les objets qu'on veut rendre persistants à un ou plusieurs objets que le système considère comme persistants par définition.

Le langage à objets PS-Algol [Atkinson83] [Cockshott84] [Atkinson85] fournit deux types pour la programmation de la persistance : Le type `pointer` et le type `table`. Le premier permet d'établir des liens entre les objets. Un objet peut contenir un pointeur à n'importe quel autre objet.

La persistance des objets résulte de leur attachement à des objets du deuxième type. Il s'agit en fait de tables de pointeurs à des objets de type quelconque. Un objet n'est pas détruit s'il fait partie d'une chaine de pointeurs qui commence dans une table. Indépendamment de son type, un objet peut donc devenir persistant. De même, une opération peut s'appliquer sur un objet sans avoir à déterminer si celui-ci est temporaire ou persistant.

PS-Algol utilise une base de données *ad-hoc* pour construire la représentation en mémoire secondaire des objets persistants. A cause du double filtre du tas du langage et de la gestion de la base de données, l'exécution d'une opération sur un objet persistant est plus lente que sur un objet temporaire.

Le besoin d'une persistance d'objets liée aux objets et non pas aux types s'impose. C'est la seule manière de la rendre indépendante d'un langage de programmation. Cette indépendance n'est possible que si la persistance est prise en compte par la fonction de gestion d'objets. Ceci a pour conséquence de la déplacer : on n'est plus dans le tas associé à un langage ou au-dessus d'une base de données, mais nettement au-dessous, au niveau de la gestion de l'espace qu'occupent les objets. Le pas suivant de l'évolution de la persistance consiste donc à descendre au niveau de la mémoire. C'est précisément ce que Thatte propose dans son modèle de mémoire persistante.

Le modèle de la mémoire persistante

Thatte propose un modèle de mémoire à un niveau pour la réalisation de la persistance. En faisant abstraction des mémoires principale et secondaire, on est amené à considérer la mémoire comme un grand et seul espace où se trouvent tous les objets. Le modèle comporte un mécanisme de gestion de la persistance dont le principe de base est le même que celui de PS-Algol : Un objet est persistant tant qu'il n'est pas récupéré par le ramasse-miettes [Thatte86].

L'idée d'une mémoire à un niveau n'est pas nouvelle : D'après Thatte, déjà dans les années 60 fut proposée une mémoire où programmes et données partageaient le seul et même contexte de désignation. La diminution de la dichotomie entre les deux niveaux de mémoire était aussi l'un des objectifs du système Multics [Organick72]. Plus récemment, une mémoire répartie à un niveau a été développée pour le système Apollo/Domain [Nelson83].

Thatte édifie sa mémoire persistante en utilisant ce qu'il appelle l''abstraction de la mémoire uniforme'. Il s'agit d'une abstraction parce que seulement sont définies les caractéristiques externes de la mémoire persistante, sans songer aux détails de l'implantation. La mémoire est uniforme car aucune distinction n'est faite entre objets temporaires et persistants.

Du point de vue du processeur qui l'utilise, une mémoire persistante possède les caractéristiques suivantes :

- Elle consiste en un ensemble d'objets comportant chacun un ou plusieurs mots de mémoire.

- Les mots d'un objet correspondent à des adresses consécutives de mémoire.
- Les mots d'un objet peuvent contenir des pointeurs (adresses de mémoire) vers d'autres objets.
- Les objets sont de taille variable.

Les caractéristiques qui mettent en évidence la persistance sont les suivantes :

- Un objet particulier de la mémoire, localisé à une adresse fixe, est la racine persistante de la mémoire. Les objets qui peuvent être atteints à partir de la racine en suivant les pointeurs sont des objets persistants. La persistance d'un objet ne dépend donc que de son appartenance à la fermeture transitive de la racine.
- Le processeur peut accéder aux objets de la mémoire si ses registres contiennent des pointeurs aux objets. Les registres du processeur définissent la racine temporaire de la mémoire : tous les objets qui appartiennent à la fermeture de la racine temporaire mais qui n'appartiennent pas à celle de la racine persistante sont des objets temporaires.
- Les objets qui ne sont ni persistants ni temporaires sont des miettes.
- Lorsque le processeur termine une exécution et que la racine temporaire change, les objets qui appartenaient à la fermeture transitive de l'ancienne racine temporaire se trouvent dans l'un des trois états suivants : soit ils appartiennent à la fermeture transitive de la nouvelle racine temporaire, soit ils ont été introduits dans la fermeture transitive de la racine persistante (par le biais d'une opération exécutée lorsqu'ils étaient temporaires), soit ils sont devenus des miettes.
- Le processeur n'a pas de moyen pour distinguer entre les objets temporaires et les objets persistants. Ils ne peuvent donc pas être traités de manière différente.

La figure Int.1.1 schématise le modèle de la mémoire persistante.

Thatte identifie cinq règles pour la préservation de l'intégrité du graphe des connexions entre objets :

1. Accès à la mémoire uniquement par le moyen des adresses logiques. Une adresse logique est représentée par la paire $<i, j>$, où j indique la position d'un mot à l'intérieur de l'objet désigné par le pointeur contenu dans le i–ième registre du processeur.
2. Vérification des adresses logiques. L'adresse logique $<i, j>$ ne dépasse pas les limites d'un objet si j n'est pas plus grand que la taille de l'objet désigné par le registre i.
3. Prévention de la contrefaçon des pointeurs.
4. Ramassage automatique de miettes.
5. Allocation contrôlée de mémoire (par 'quotas', par exemple).

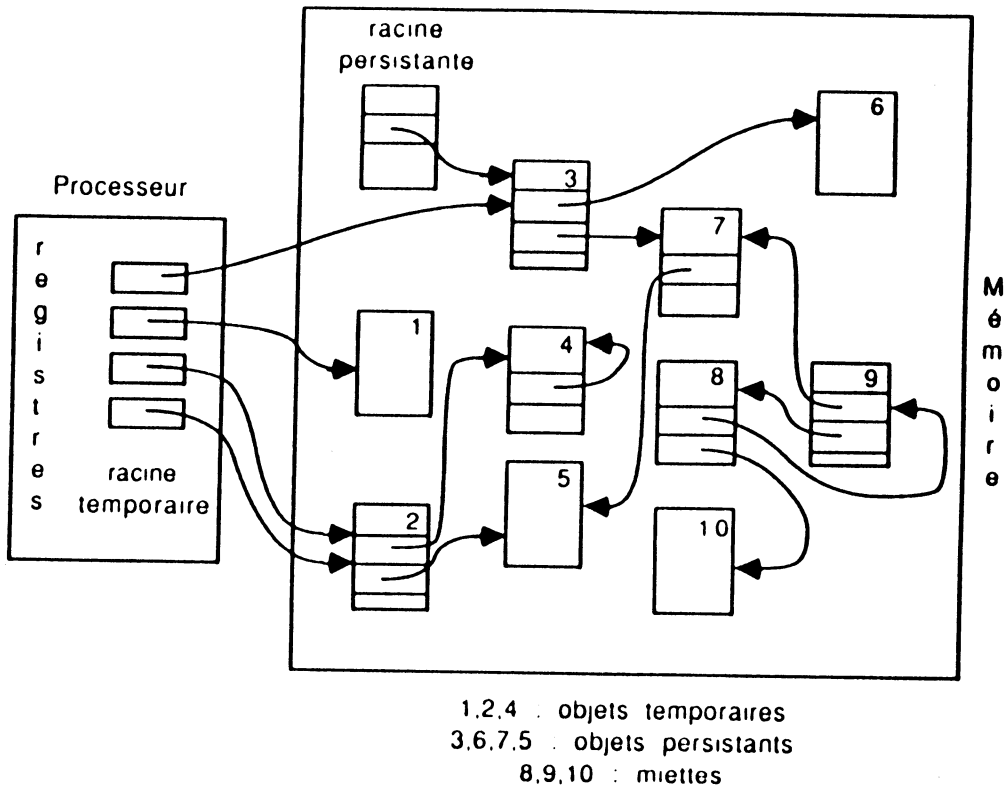


Figure Int.1.1 - Modèle de la mémoire persistante

Les trois premières conditions sont garanties si les programmes qui utilisent la mémoire sont écrits en langages de haut niveau et si la probabilité de programmer en assembleur est réduite. Sans les deux dernières conditions, le système peut se voir obligé de s'arrêter à cause d'un manque d'espace.

Les trois considérations suivantes doivent être prises en compte lors de la mise en oeuvre du modèle :

1. La mémoire persistante est en même temps un support d'exécution et de conservation. Une coopération entre les deux niveaux de mémoire s'avère donc nécessaire pour son implantation. De ce point de vue, la mémoire persistante se présente comme une mémoire virtuelle d'objets.

2. Tous les utilisateurs peuvent accéder aux objets de la mémoire persistante. Son implantation doit donc viser la mise en place de mécanismes de partage d'objets et de contrôle de concurrence.

3. Des modifications sur les objets étant possibles, la mémoire doit pouvoir les conserver de manière persistante. Bien que le modèle prenne en compte l'intégrité de la structure des connexions entre les objets, rien n'est dit sur la résistance aux pannes et sur la reprise. L'implantation doit en tenir compte.

La répartition

L'influence de la répartition sur la gestion de la persistance concerne deux aspects : la cohérence et la désignation de données.

L'accès aux données peut se faire sur un site autre que celui où se trouve leur représentation en mémoire secondaire. Dans ce cas, le maintien de la cohérence exige que les éventuelles modifications soient amenées sur le site de résidence de cette représentation. Le partage des données doit aussi être contrôlé, en particulier lorsque deux représentations en mémoire principale de la même donnée se trouvent sur deux sites différents.

Lorsque plusieurs représentations en mémoire secondaire sont autorisées pour la même donnée, il est nécessaire de contrôler la manière dont les modifications sur une copie sont faites et propagées sur les autres copies. Il existe pour ceci plusieurs techniques, comme celle des copies primaires et secondaires [Popek85] et celle du vote [Gifford79].

En termes généraux, il existe deux paradigmes de contrôle de cohérence dans un environnement réparti : la sérialisation de processus et la synchronisation de processus. Le premier s'applique lorsque les données dont la cohérence doit être préservée sont partagées. Il s'agit du type de cohérence qu'assurent les transactions. Parmi les systèmes transactionels récents, on peut citer Argus [Liskov88] et Camelot [Spector88].

La synchronisation de processus distants est nécessaire lorsque ces processus travaillent en coopération, notamment pour la manipulation de données dupliquées. C'est aussi le cas des processus qui, à la suite d'une défaillance d'un autre, prennent la relève et finissent des calculs critiques ou qui portent sur des ressources en exclusion mutuelle. Les processus interagissent par échanges de messages plutôt que par partage de mémoire. Le système Isis permet de mettre en oeuvre des applications qui ont besoin de ce type de répartition [Birman87].

Dans les systèmes répartis à objets, tout objet peut référencer des objets qui se trouvent sur des sites différents. Les mécanismes de désignation doivent donc tenir compte qu'il est nécessaire de connaître non seulement les adresses des objets mais aussi leurs sites de résidence. Les systèmes de gestion de fichiers répartis maintiennent des tables qui réalisent la correspondance <nom de fichier[2], site de résidence>.

La gestion de cette désignation intersites se complique lorsque les objets ou fichiers migrent. En effet, les références sur un objet ou le nom d'un fichier ne sont plus valides si l'objet ou le fichier changent de site de résidence. On doit pouvoir compter sur des mécanismes qui permettent soit de mettre à jour ces informations, soit de rediriger l'utilisateur vers le nouveau site de résidence.

---

2. On entend ici par 'nom de fichier' soit l'identificateur interne d'un fichier, soit son nom symbolique.

Enfin, la distribution amène pour le ramasse-miettes le problème des cycles répartis de miettes : un site seul ne peut pas déterminer qu'il y a un cycle. Le ramasse-miettes exige donc la colaboration de tous les sites pour les éliminer [Dijkstra83].

2. Le système Guide

Guide (*Grenoble Universities Integrated Distributed Environment*) est un projet de recherche commun au Laboratoire de Génie Informatique (IMAG, Grenoble) et au Centre de Recherche Bull (Grenoble). Son objectif est la conception et la réalisation d'un système d'exploitation réparti sur des postes de travail individuels et des serveurs spécialisés. Dans un premier temps, Guide est un système expérimental qui doit servir de banc d'essai pour des recherches sur les systèmes répartis [Balter86] [Guide-R1]. Une partie des travaux du projet Guide est menée dans le cadre du projet ESPRIT 834, Comandos (*Construction and Management of Distributed Office Sytems*) [Comandos87].

Le trait le plus important du système Guide est celui de l'**intégration**. Celle-ci a lieu dans deux directions :

1. Intégration de composants : Chaque poste de travail est considéré comme un composant d'un système d'exploitation global réparti. Les fonctions liées à la répartition ne sont pas surajoutées à un système déjà existant mais prises en compte dès l'origine.

2. Intégration de concepts en 'virtualisant' les ressources : Les ressources d'exécution et de conservation sont rendues virtuelles afin de décharger les utilisateurs de la nécessité de connaître leurs détails. Cette 'virtualisation' est faite de manière à favoriser une vue intégrée des composants du système.

La mise en oeuvre de cette intégration a été possible par l'incorporation des aspects de conservation de l'information au pouvoir d'abstraction des modèles à objets. De cette manière, l'objet, entité de base du système, en constitue l'unité d'abstraction, d'exécution et de conservation. Ce triple rôle ressort des trois modèles en termes desquels a été conçu le système. Ces modèles sont :

1. Le modèle de données. Il permet aux usagers d'abstraire, en utilisant des objets, la nature des données de leurs applications. Ce modèle est mis en oeuvre par le moyen d'un langage particulier, nommé 'langage Guide'.

2. Le modèle d'exécution. Il définit les environnements d'exécution du système. Un environnement d'exécution est constitué d'un ensemble d'objets et d'un ensemble de processus qui opèrent sur eux.

3. Le modèle de la mémoire. Il détermine les caractéristiques de l'espace d'exécution et de conservation qui comporte tous les objets du système. La mémoire du système Guide constitue une réalisation du modèle de la mémoire persistante de Thatte.

La superposition des modèles d'exécution et de mémoire met à la disposition des concepteurs d'applications une machine virtuelle multiprocesseurs dans laquelle le parallélisme est apparent et la distribution est cachée. Cette machine réalise donc les fonctions de la gestion d'objets. L'interface d'accès à la machine virtuelle est constituée par les instructions du langage Guide.

Certaines des fonctions de la gestion d'objets n'ont pas été intégrées comme fonctions de machine virtuelle. Il s'agit de fonctions non essentielles pour le bon déroulement de système et qui, de surcroît, ne sont pas utilisées par toutes les applications. Elles constituent des services réalisés au-dessus de la machine virtuelle et qui utilisent l'interface de celle-ci. La désignation symbolique d'objets, la gestion de types et classes et la gestion de versions sont trois exemples des services du système Guide.

3. Le travail réalisé

La définition de la machine virtuelle de Guide en termes de deux modèles qui se superposent a établi de manière naturelle deux ensembles différents de fonctions de gestion d'objets. Nous avons choisi l'ensemble correspondant au modèle de la mémoire. Les objectifs principaux de ce travail de thèse sont donc la conception et la mise en oeuvre des fonctions de gestion de la mémoire d'objets de Guide.

Cette mémoire d'objets est en même temps le support d'exécution et de conservation du système. La réalisation de ces deux fonctions est la cause du découpage naturel de la mémoire en deux niveaux :

1. Le support des objets en mémoire principale, ou **Mémoire Virtuelle d'Objets** (MVO), qui permet l'exécution d'opérations sur les objets.
2. Le support des objets en mémoire secondaire, ou **Mémoire Permanente d'Objets** (MPO), qui réalise la conservation des objets.

La gestion de la mémoire d'objets comporte donc des fonctions de gestion de la mémoire principale (ou virtuelle) et des fonctions de gestion de la mémoire secondaire. Il existe une liaison très forte entre ces groupes de fonctions.

D'autres objectifs de la thèse, également importants, sont la conception et la mise en oeuvre de deux des fonctions de la gestion d'objets qui n'ont pas été intégrées dans la machine virtuelle. Il s'agit des fonctions de désignation symbolique d'objets et de gestion des versions des objets. L'association de la gestion de la mémoire à la désignation symbolique et à la gestion de versions est due au fait qu'il existe des fonctions similaires dans les systèmes de gestion de fichiers, qui ont constitué notre premier domaine d'étude et qui ont eu, au moins dans l'aspect de gestion de la mémoire secondaire, une grande influence sur la manière dont les systèmes à objets assurent la persistance 'classique'.

C'est à cause de cette analogie avec les systèmes de gestion de fichiers qu'on s'est habitué, au sein du projet Guide, à identifier le terme 'fonctions de gestion d'objets'

avec les fonctions de gestion de la mémoire et même avec celles de gestion de la mémoire secondaire. Ceci obéit aussi à d'autres raisons : D'une part, il n'est pas toujours facile de séparer la gestion de la mémoire principale de la gestion des entités du modèle d'exécution, étant donné l'étroit rapport qui existe entre elles. D'autre part, la gestion de la mémoire secondaire a été développée de façon assez indépendante du reste du système, et on la voit parfois comme la fonction d'un serveur de 'morceaux' de mémoire qui pourrait éventuellement être utilisée par d'autres systèmes comme support de conservation.

Notre premier effort dans cette thèse est de faire ressortir la liaison existante entre la MVO et la MPO et de donner une vision uniforme de la mémoire comme réalisation du troisième modèle du système. La mémoire de Guide est une mémoire d'objets répartie qui, par l'intégration d'un ramasse-miettes, est aussi une mémoire persistante. Ceci éclaircit un autre point assez confus au sein de l'équipe Guide : la MPO est le support de conservation des objets mais elle n'est pas leur seul support de persistance. Celle-ci résulte des interactions entre les *deux* mémoires. Persistance *n'est pas seulement* conservation. La persistance concerne aussi le fait que les modifications faites sur la représentation d'un objet en MVO soient incorporées à la représentation en MPO. Cette incorporation ne se fait pas, comme dans la persistance classique, par l'initiative des applications mais lorsque la gestion de la MVO a besoin de l'espace qu'occupent les objets non utilisés depuis longtemps pour en accueillir d'autres. La persistance concerne encore le ramassage de miettes, qui peut être mis en oeuvre aussi bien en MVO qu'en MPO (cf. section 4.4). Ces considérations indiquent que la MVO est aussi concernée par la persistance des objets.

Dans le système Guide, les objets temporaires n'existent que pendant l'exécution d'une opération sur un autre objet. Ils sont mis en oeuvre dans la pile du processus qui exécute cette opération. Ainsi, ils meurent à cause du mécanisme du modèle d'exécution qui détruit le contenu de la pile d'un processus à la mort de celui-ci. Les objets temporaires sont donc détruits en même temps qu'un processus et non pas parce qu'ils deviennent des miettes à la fin du processus et qu'on les ramasse ensuite.

En conséquence, la gestion de la mémoire ne concerne que les objets qui ont une représentation en mémoire secondaire, donc persistants dans le sens classique. Etant donné que tout objet de la mémoire possède une représentation en MPO et que, d'après les politiques de gestion d'espace de la MVO, tout objet non utilisé depuis un certain temps est recopié en MPO, il est possible de réaliser un ramasse-miettes qui récupère l'espace des objets en MPO. Le ramassage d'objets en MVO est aussi possible, mais il doit être forcément suivi de la destruction de leur représentation en MPO.

Les premières fonctions de gestion de la mémoire d'objets sont la création et la destruction des représentations en MVO et en MPO. La destruction d'une représentation en MVO consiste à récupérer son espace en mémoire principale (virtuelle) après l'avoir copié en MPO, si des modifications ont été faites. La création d'une représentation en MVO consiste à lui réserver de l'espace en mémoire principale (virtuelle) avant de charger le contenu de la représentation en MPO. Ainsi, dans un premier temps, l'interface entre la MVO et la MPO comporte les fonctions de chargement et de

rangement. Le chargement ne se fait que lorsqu'il est nécessaire d'exécuter une opération sur l'objet et qu'il n'a pas de représentation en MVO. On appelle cette situation **défaut d'objet**. Le rangement ne se fait que lorsque l'espace de mémoire de la représentation en MVO doit être récupéré. La mémoire d'objets de Guide est donc une mémoire virtuelle.

La création d'une représentation en MPO constitue la vraie création de l'objet : on lui réserve de l'espace de conservation et on lui affecte un identificateur. L'objet est déjà persistant dans le sens classique. La destruction d'une représentation en MPO constitue la vraie mort de l'objet : elle n'est réalisée que par le ramasse-miettes. L'objet est ainsi persistant dans le sens du modèle de Thatte.

Une autre fonction de la mémoire consiste à limiter à *un* le nombre de représentations en MVO de chaque objet. On appelle cette fonction le 'verrouillage d'objets'. Le but de cette restriction est de forcer les environnements d'exécution à s'étendre sur la mémoire principale où se trouve la représentation de l'objet auquel ils veulent accéder. Plusieurs environnements peuvent ainsi partager la représentation en MVO de cet objet. Un objet ne peut pas être libéré (déverrouillé) tant qu'il y a un environnement qui l'utilise.

Les dernières fonctions de la gestion de la mémoire sont celles de désignation et de localisation des objets. On utilise un seul espace d'adressage : celui des références. Une référence sert à désigner un objet indépendamment de son type et du niveau de mémoire ou du site où il se trouve. La références donnent aussi le moyen de le localiser. Si l'information d'une référence n'est pas valide, à cause par exemple d'une migration, un mécanisme de localisation permet de retrouver l'objet et de mettre à jour la référence.

La première partie de ce travail a donc consisté à proposer un modèle de la mémoire d'objets, avec les deux niveaux et les fonctions qu'on vient de décrire. Puisque la machine virtuelle la cache, la répartition n'apparaît que lors de la mise en oeuvre du modèle. Ce n'est en effet qu'au moment de la présentation de cette mise en oeuvre qu'on parle des 'sites Guide'. Le modèle de la mémoire, de même que les deux autres qui constituent le système Guide, a été mis en oeuvre au-dessus du système Unix™ V.2.

La deuxième partie du travail a consisté en la conception et mise en oeuvre des fonctions de désignation symbolique et de gestion de versions. On a fait de ces fonctions deux services du système Guide, programmés en langage Guide comme des applications. Bien que leurs fonctions soient différentes, les services ont été réalisés de manière semblable : une classe d'objets qui met en oeuvre une désignation particulière. Il s'agit de la désignation par noms symboliques et d'une désignation assez souple pour les objets qui constituent l'ensemble des versions d'un autre objet.

™ Unix est une marque déposée d'*ATT Bell Laboratories*.

Le service de désignation symbolique fournit des noms similaires à ceux des fichiers d'Unix, avec des répertoires, des racines relatives à un répertoire, des chemins d'accès et des parcours de chemin. Une classe définit les répertoires, les noms étant, comme dans le cas des fichiers d'Unix, des chaînes de caractères.

Le service de gestion de versions se veut un outil général qui permet de bâtir au-dessus des gestions des versions adaptées aux besoins des applications. Il permet de manipuler les versions d'un point de vue à la fois temporel et fonctionnel. En effet, les versions maintiennent des rapports non seulement selon la manière dont les unes sont derivées à partir des autres, ce qui donne un arbre de dérivation, mais aussi selon le moment de leur création, ce qui permet de construire un historique.

Tout objet de Guide peut avoir un ou plusieurs noms symboliques ou donner lieu à un nombre quelconque de versions.

4. Plan de la thèse

La these est divisée en deux parties. La première présente un état de l'art sur deux des types de persistance identifiés dans la section précédente et situe Guide par rapport aux systèmes analysés. En plus de la persistance, deux autres aspects ont été considérés : la gestion de la mémoire, principale et/ou secondairte, et le contrôle de concurrence pour le partage de données. On a pris en compte des réalisations réparties et centralisées ; on a donc étudié les implications que la répartition a sur la persistance, la désignation et le partage. Cette première partie comporte deux chapitres : Dans le premier, on analyse des systèmes qui garantissent une persistance classique à leurs données. Le deuxième considère les tendances actuelles de traitement de la persistance, c'est-à-dire, les réalisations du modèle de la mémoire persistante de Thatte ou des systèmes qui offrent les moyens pour la mise en oeuvre de ce modèle.

La deuxième partie de la thèse présente le modèle de la mémoire d'objets du système Guide et sa réalisation. Le développement des services de désignation symbolique et de gestion de versions est également décrit. Cette partie comporte cinq chapitres : D'abord, une introduction à Guide présente quelques détails sur les deux autres modèles du système et leur influence sur celui de la mémoire d'objets. Le chapitre suivant introduit les détails du modèle de la mémoire : les deux niveaux, les fonctions de chaque niveau et de l'interface, la structure interne, le cycle de vie des objets et quelques idées sur le ramasse-miettes. Le troisième chapitre de cette partie se concentre sur la conception des services de désignation symbolique et de gestion de versions.

Les deux derniers chapitres décrivent les réalisations : du modèle de la mémoire d'abord, des services ensuite. C'est dans le chapitre de la réalisation du modèle qu'est rendue visible la répartition du système. Les sites du réseau sont caractérisés ainsi que le système support qui a permis le développement de Guide. En revanche, la répartition n'apparaît pas dans le dernier chapitre. En effet, les services étant conçus en terme du modèle de données, elle est à nouveau cachée par la machine virtuelle. C'est aussi pour

cette raison que la taille du chapitre est très réduite.

La conclusion montre l'accomplissement des objectifs : On a réalisé une mémoire virtuelle d'objets qu'on veut aussi mémoire persistante. On a mis en oeuvre les services de désignation symbolique et de gestion de versions. On présente enfin les enseignements acquis au cours de cette étude.

Première partie : Des systèmes de gestion de fichiers aux systèmes de gestion d'objets

Avant-propos

L'objectif de cette première partie est d'étudier la manière dont certains systèmes réalisent la persistance des informations. Nous nous intéressons à deux des types de persistance présentés dans l'introduction, à savoir, la persistance 'classique' et les tendances actuelles de traitement de la persistance. Ces dernières comprennent les réalisations du modèle de la mémoire persistante de Thatte et les systèmes qui offrent les moyens pour la mise en oeuvre de ce modèle.

D'autres aspects considérés, plus ou moins liés à la persistance, sont la gestion de la mémoire, qui comporte le ramassage de miettes, et le contrôle de concurrence, nécessaire pour autoriser le partage des entités. Le terme 'entité' englobe les objets, les fichiers et les relations et n-uplets d'une base de données.

Finalement, nous voulons situer Guide par rapport aux autres systèmes. On ne rentre cependant pas dans les détails de conception et de réalisation du système Guide car ceci constitue l'objectif de la deuxième partie de ce travail.

On a classifié les systèmes selon que le degré d'intégration existant entre les deux niveaux de mémoire permet de réaliser l'une ou l'autre des définitions de persistance. Nous distinguons ainsi :

1. Les systèmes de gestion de fichiers des systèmes d'exploitation classiques, les serveurs de fichiers et les systèmes de gestion de bases de données qui utilisent l'un des trois modèles de données traditionnels. Dans ces systèmes, les deux organisations de mémoire sont différentes et, surtout, complètement indépendantes. En plus, dans le cas des systèmes de fichiers, l'unité d'échange n'est pas le fichier mais une chaîne de caractères quelconque.

2. Les serveurs de segments (morceaux) de mémoire que des systèmes spécialisés utilisent pour la réalisation de mémoires persistantes. Ces segments ne sont pas, comme dans le cas des systèmes de gestion de fichiers, manipulés par le moyen de noms symboliques mais par le biais d'identificateurs uniques. Le segment de mémoire constitue l'unité d'échange entre les deux niveaux de mémoire et est utilisé par le système spécialisé pour la mise en oeuvre d'un objet. Ces caractéristiques établissent une intégration plus marquée entre le serveur et le système spécialisé. C'est, en effet, le système qui attend du serveur une gestion de la mémoire en segments. Le serveur ne réalise aucune action en ce qui concerne le contenu des segments.

3. Les systèmes qui réalisent le modèle de mémoire persistante de Thatte. En général, un système de cette catégorie fait partie d'un autre système qui utilise la mémoire persistante pour la mise en oeuvre d'un modèle de données ou pour satisfaire directement les besoins de ses applications.

Certains des systèmes analysés, en particulier ceux qui appartiennent aux deux premières catégories, sont particulièrement riches en organisation interne de la mémoire

secondaire et en mécanismes de contrôle de concurrence. Ces caractéristiques acquièrent un intérêt particulier lorsque les systèmes sont mis en oeuvre de manière répartie. Les aspects où la distribution influence la gestion de la persistance sont aussi analysés.

Dans les études comme celle-ci, il est impossible d'être exhaustif. Les systèmes analysés ont été choisis parce qu'ils constituent de bons exemples de mise en oeuvre de la persistance et parce que la documentation disponible sur eux nous a permis de comprendre certains détails qui ne relèvent pas de la simple description de leurs objectifs immédiats.

Les systèmes analysés sont les suivants :

1. Le *Network File System* (NFS), développé par Sun Microsystems, Inc. [Lyon84] [Sandberg86].
2. Le système Andrew, développé par ITC (Information Technology Center), une organisation de collaboration entre IBM et Carnegie-Mellon University [West85] [Morris86] [Satyanarayanan84]. [Satyanarayanan85]
3. Le SGBD relationnel Oracle développé et commercialisé par Oracle Corporation [Oracle85].
4. Le serveur d'objets ObServer, développé au Computer Science Department, Brown University [Skarra86a] [Hornik87].
5. Deux réalisations d'une memoire d'objets pour le système Smalltalk-80 : La mémoire LOOM, développée au Xerox Palo Alto Research Center [Kaehler86], et le gestionnaire réparti d'objets, développé par D. Decouchant au Laboratoire de Génie Informatique, IMAG, Grenoble [Decouchant87].
6. L'implantation que S. Thatte réalise de son propre modèle de mémoire persistante, développée au Artificial Intelligence Laboratory, Computer Science Center, Texas Instruments, Inc. [Thatte86].

Cette étude comporte deux chapitres : l'un pour les aspects de la persitance classique, l'autre pour les tendances actuelles. A la fin de chaque chapitre, les systèmes analysés sont comparés entre eux et avec le système Guide.

Un aspect qui n'est traité dans aucun des systèmes analysés est celui de la gestion de versions. Il s'agit, en fait, d'une caractéristique orthogonale à la persistance mais qui fait partie de la gestion d'objets (ou de fichiers). De ce point de vue, un système peut considérer une version de deux manières :

1. Comme le support de validation des modifications (*commit*).
2. Comme le resultat de la gestion du temps faite par les applications du système.

Dans le premier cas, les modifications ne se réalisent pas sur l'objet (ou fichier) mais sur une copie créée à cet effet. La copie constitue une nouvelle version. La validation des modification consiste à remplacer l'objet (ou fichier) par la copie modifiée. Ce type de versions est fourni par le serveur de fichiers Felix [Fridrich81].

Tout système qui supporte le deuxième type de versions doit prendre en compte la manière et la fréquence de production de celles-ci. Ce sont les applications qui déterminent la manière de produire les versions. Une application peut considérer, par exemple, que les versions des objets d'un type particulier sont constituées par des objets de type différent. C'est le cas des versions successives qui reflètent les étapes (fonctionnelle, logique, électrique, etc.) propres aux objets manipulés par les applications de CAO [Rieu86].

Toute application détermine également le moment de production des versions. Dans [Bui Quang86], trois types de versions sont identifiés selon la fréquence dans laquelle elles se succèdent dans le temps. Les versions 'manuelles' sont obtenues chaque fois que l'application le demande explicitement. Les versions 'successives' sont produites à chaque exécution d'une opération choisie par l'application. Enfin, les versions 'périodiques' sont créées à chaque expiration d'une période dont la durée a été déterminée par l'application.

Le SGBD Geode [Pucheral88] offre un mécanisme de versions qui répose sur les notions d'identificateur physique et logique. Un seul et même identificateur logique autorise l'accès à un ensemble d'objets ayant chacun son identificateur de version et son identificateur physique. Il s'agit donc d'un mécanisme de base qui permet de manipuler les versions tout en laissant aux applications la gestion sémantique qui leur est associée.

Un aspect intéressant du SGBD Encore [Zdonik86] est précisément la gestion de versions. Afin de gérer les versions au niveau des objets, deux types particuliers sont introduits dans le système : `historic` et `version-set`. L'ensemble des propriétés du type `historic` comporte `previous-version` et `next-version`, la deuxième propriété étant multi-valuée (cf. section 2.1.3). Le type `version-set` est utilisé pour regrouper les versions d'un objet. Un type étant lui-même un objet, il peut aussi avoir de versions. Un mécanisme d'exceptions permet aux instances d'un type d'être traitées comme des instances d'une version quelconque du type [Skarra86b].

Etant destiné à supporter des applications de CAO, domaine où la gestion de versions joue un rôle déterminant, le SGBD Etic [Fauvet88] permet de caractériser plus profondément les versions des objets de ses applications. Une version se distingue par son contexte de création, par son état, par ses propriétés et par les rapports qu'elle maintient avec les versions dont elle a été dérivée. Le système définit en outre des primitives de manipulation et de contrôle de versions.

1. Gestion classique de la persistance

La persistance classique est celle des informations qui survivent aux procédures ou processus qui les manipulent. Cette survie leur est garantie par les systèmes de gestion de fichiers ou de bases de données, bien que ceci ne constitue pas la seule fonction de ces systèmes. L'unité de gestion d'information en mémoire principale ne correspond pas à l'unité de gestion en mémoire secondaire. En conséquence, les traitements qu'une information reçoit ne sont pas uniformes dans les deux niveaux.

Aucun ramassage automatique d'information n'est réalisé par ces systèmes. En réalité, ils ne connaissent pas les 'miettes' telles qu'elles sont définies et traitées dans les systèmes de programmation symbolique ou à objets. Les utilisateurs sont responsables de la destruction de leurs informations, sauf si l'administration de leur système instaure une politique de gestion d'espace selon laquelle certaines informations sont effacées automatiquement de temps en temps.

1.1 Systèmes de gestion de fichiers

La gestion de fichiers est une fonction de base des systèmes d'exploitation [Krakowiak85]. Cependant, elle tend de plus en plus à se détacher de ces systèmes. Ce phénomène a pour cause l'adoption des architectures réparties qui prônent la séparation entre les postes de travail et les machines 'serveur' d'un réseau. Soit intégrés encore au système, soit mis en oeuvre par un serveur (ou ensemble de serveurs) indépendant, les systèmes de fichiers continuent à fournir le support de persistance classique. Les caractéristiques qui marquent leur évolution ne portent pratiquement que sur la répartition et les conséquences qui découlent de celle-ci.

On a choisi NFS et le système Andrew parce que leurs caractéristiques, en ce qui concerne la mise en oeuvre du partage comme conséquence de la répartition, sont assez différentes. Il y a évidemment beaucoup d'autres systèmes qui réalisent des fonctions semblables. On peut trouver dans [Svobodova84] une bonne synthèse des aspects principaux d'un vaste ensemble de systèmes de fichiers. Une analyse similaire, mais prenant en compte uniquement des systèmes de fichiers construits à partir d'Unix, est réalisée dans [Wupit83]. Enfin, dans [Scioville86], apparaît une étude comparative sur quelques systèmes de gestion de fichiers distribués.

1.1.1 NFS

Le *Network File System* (NFS) de Sun est un ensemble de services qui permettent aux systèmes Unix 4.2 BSD et Unix V.2 de partager des fichiers sur un réseau local non homogène [Lyon84] [Sandberg86]. Les plus importants de ces services sont les suivants :

1. Le service d'accès aux fichiers, appelé aussi 'service NFS'. Il permet l'accès et le partage des fichiers stockés sur les différents sites du réseau.
2. Le service de verrouillage. Il bloque et débloque à la demande les fichiers stockés sur les différents sites du réseau.
3. Le service de montage (*mount*). Il se charge de la mise en place de l'espace de désignation de fichiers.

Les services sont indépendants et chacun comporte son propre protocole de communications. A l'exception du service NFS, qui a été intégré au noyau d'Unix, ils ont tous été réalisés comme des ensembles de processus-utilisateur. Sauf indication du contraire, la description qui suit concerne uniquement le service NFS.

1.1.1.1 L'espace de désignation des fichiers

L'administrateur d'un site Unix peut autoriser le partage d'une sous-hiérarchie quelconque de la hiérarchie de fichiers du site. La primitive `export` du service de montage lui permet d'indiquer, d'une part, les sous-hiérarchies qu'il veut partager et, de l'autre, les sites auxquels il autorise ce partage.

Si les utilisateurs du site *x* veulent accéder aux fichiers de l'une des sous-hiérarchies du site *y*, l'administrateur de *x* doit vérifier que son site a été autorisé pour réaliser le partage. La primitive `import` du service de montage est exécutée à cette fin.

Pour que les utilisateurs du site *x* puissent accéder effectivement aux fichiers d'une sous-hiérarchie du site *y*, il faut que cette sous-hiérarchie soit liée d'une certaine manière à la hiérarchie de fichiers de *x*. L'idée de base est de rendre transparent l'accès, comme si la sous-hiérarchie de *y* appartenait à la hiérarchie de *x*.

De manière similaire au montage classique d'Unix, l'administrateur du site *x* choisit un fichier de sa hiérarchie et monte sur lui la sous-hiérarchie du site *y*. Désormais, toute référence à ce fichier, qu'on appelle 'point de montage', ne permet plus l'accès à son contenu mais à la racine de la sous-hiérarchie de *y*.

Le montage d'une sous-hiérarchie distante sur un point de montage local est réalisé par la primitive `mount` du service de montage. Dans l'*inode* du point de montage sont placés l'identificateur du site de résidence de la sous-hiérarchie montée et l'identificateur de la racine de cette sous-hiérarchie. Ce deuxième identificateur est local au site de résidence.

La figure 1.1 montre le partage de sous-hiérarchies entre quatre sites du réseau. On peut remarquer que ce partage établit une relation client-serveur entre les sites : un site qui 'exporte' une sous-hiérarchie est le serveur du site qui l''importe'. Ce deuxième site est donc le client. Un site peut être à la fois client et serveur.

La primitive `lookup` du service NFS permet de réaliser le parcours du chemin d'accès à un fichier. Elle reçoit comme paramètres l'identificateur d'un *directory* et le nom d'un des fichiers catalogués dans ce *directory*. Le résultat est l'identificateur du

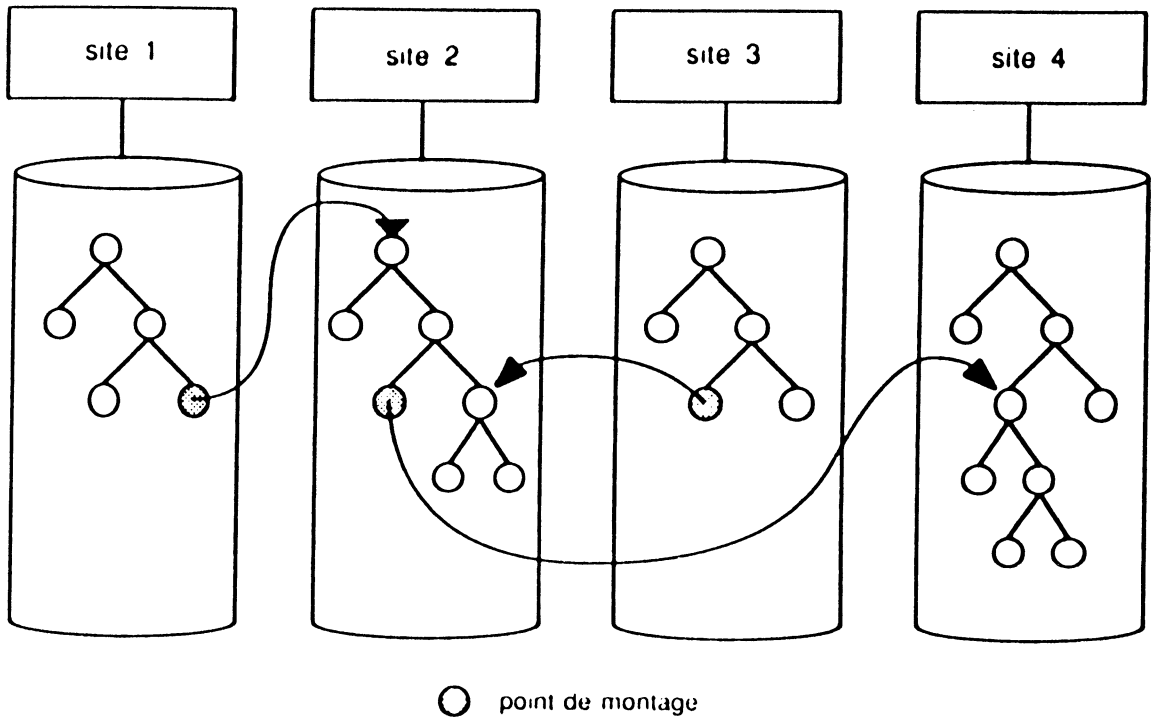


Figure 1.1 - Espace de désignation de fichiers sous NFS

fichier dont le nom a été fourni en paramètre.

Le parcours est réalisé par une suite d'appels à `lookup` appliqués sur les noms compris dans le chemin. Chaque appel reçoit comme paramètres le résultat de l'appel précédent et le nom suivant dans le chemin. La suite commence à partir du *directory* de travail (*working directory*) et se termine à l'obtention de l'identificateur du fichier dont le nom est le dernier du chemin. Un *inode* est établi pour conserver cet identificateur.

Si un point de montage est trouvé dans le chemin, les appels ne s'exécutent plus sur le site local mais sur celui indiqué dans l'*inode* du point de montage. Le site local envoie des requêtes pour demander l'exécution de *chaque* appel de la suite. A la fin du parcours, un *inode* est établi sur le site local pour conserver l'identificateur du fichier et l'identificateur du site où celui-ci se trouve réellement.

On peut remarquer que, sur un même site, deux sortes d'*inode* peuvent exister : celui qui contient les informations d'un fichier local, qui ne présente aucune différence avec un *inode* classique Unix, et celui qui contient les informations d'un fichier distant.

1.1.1.2 Le client

Un site du réseau se comporte comme un client lorsqu'il accède aux fichiers des sous-hiérarchies distantes qu'il a montées.

Le noyau du système Unix est augmenté d'une interface qui banalise les différences entre les divers fichiers auxquels un client peut accéder. Cette interface, dite 'virtuelle', utilise deux structures de données identiques pour grouper les primitives d'accès aux fichiers : l'une pour les fichiers locaux et l'autre pour les fichiers distants[3]. Le nombre et les noms des primitives sont les mêmes dans chaque structure.

Les processus des utilisateurs font des appels-système Unix. Les appels-système concernant les accès aux fichiers ont été modifiés de façon à appeler les primitives de l'interface virtuelle. La structure de données correcte est choisie en fonction des informations contenues dans les *inodes*.

Les primitives sur les fichiers locaux sont les opérations classiques Unix d'accès aux fichiers. Une primitive sur un fichier distant consiste en un appel distant. Dans ce cas, une requête est envoyée au site dont l'identificateur se trouve dans l'*inode* correspondant. Le premier paramètre d'une telle requête est l'identificateur du fichier distant, qui se trouve aussi dans cet *inode*. On peut constater que la primitive *lookup* suit exactement ce schéma d'exécution.

1.1.1.3 Le serveur

Un site du réseau se comporte comme un serveur lorsqu'il donne suite aux requêtes envoyées par les clients.

Les requêtes sont traitées par un processus-noyau d'Unix, appelé 'le serveur NFS'. Le premier paramètre de la requête permet au serveur de retrouver l'*inode* du fichier visé. Selon le contenu de celui-ci, il choisit une structure de données de l'interface virtuelle afin d'exécuter la primitive correspondante à la requête. Le résultat de l'exécution est envoyé au client.

Le serveur NFS ne garde jamais de traces sur sa liaison avec un client. Les informations qui arrivent dans chaque requête lui permettent de compléter l'opération demandée. Ceci implique qu'il doit réaliser la validation (*commit*) des modifications lors de chaque requête d'écriture *avant* d'envoyer la réponse au client. Il faut valider non seulement les pages modifiées mais aussi les *inodes* des fichiers concernés. En

---

3. En fait, si sur le site ont été montés localement des hiérarchies de fichiers de type différent à Unix (Ultrix ou VMS™, par exemple), une autre structure de données identique est utilisée pour grouper les primitives d'accès aux fichiers de ce type [Sandberg86]. VMS est une marque déposée de *Digital Equipment Corp.*

conséquence, la gestion des *buffers* du système Unix a dû être modifiée.

Le contrôle de concurrence est basé sur le verrouillage de fichiers au niveau des requêtes. Le serveur NFS verrouille implicitement les fichiers concernés par une requête. Le type de verrouillage est celui d'Unix : 'plusieurs-lectures/une-écriture'. Les verrous sont maintenus jusqu'à la fin de l'exécution de la primitive correspondante à la requête. Pour permettre un niveau de contrôle de concurrence plus riche que le niveau requête, le service de verrouillage offre aux clients des primitives pour l'acquisition et le relâchement explicites de verrous.

Le site d'un serveur NFS peut être le client d'un autre serveur NFS mais il ne sera jamais un intermédiaire entre un client et un serveur NFS. Face à l'éventualité d'introduire des cycles dans la hiérarchie des fichiers auxquels les utilisateurs d'un site peuvent accéder, le service NFS a été conçu de façon à *ne jamais traverser* les points de montage trouvés lors du parcours d'un chemin d'accès.

1.1.1.4 Mise en oeuvre

Un mécanisme d'appel de procédure à distance (*remote procedure call* ou RPC) a été implanté pour les communications entre les processus des utilisateurs des clients et les processus qui mettent en oeuvre les services du système NFS. Ce RPC est indépendant du protocole de transport (UDP/IP) et du système Unix.

Un protocole de présentation a été défini pour autoriser la connexion de machines hétérogènes dans un réseau NFS. Ce protocole, nommé *External Data Representation* (XDR), permet de définir une représentation externe pour chaque type de données et de spécifier les primitives de traduction entre les représentations interne et externe.

1.1.2 Le système Andrew

L'objectif du système Andrew est de permettre le partage d'information dans le cadre du réseau d'un grand environnement universitaire [West85] [Morris86] [Satyanarayanan84] [Satyanarayanan85]. Le système a été mis en oeuvre sur deux types de machines : les postes de travail et les sites serveurs. Un utilisateur peut accéder, depuis un poste quelconque, à un fichier rangé dans la mémoire secondaire de n'importe quel site serveur. Les sites serveurs gèrent la totalité des fichiers partagés par tous les postes de travail.

Le système des postes de travail et des sites serveurs est Unix 4.3 BSD. Les fichiers partagés sont donc des fichiers Unix. La topologie du réseau vise à maintenir une bonne performance de travail dans un réseau prévu pour comporter entre 5000 et 10000 postes.

La gestion des fichiers partagés est réalisée comme un ensemble de processus-noyau Unix coopérants résidant chacun dans un site serveur. Ce système réparti est appelé *Vast Integrated Computing Environment* (Vice). Sur chaque poste de travail s'exécute

un processus-noyau Unix, appelé Venus, qui gère tous les échanges entre le poste et Vice. Les applications des utilisateurs des postes ne peuvent avoir accès aux fichiers de Vice qu'à travers Venus.

Le partage de fichiers sous Andrew se base sur le mécanisme des caches de fichiers. Lorsqu'une application d'un poste de travail veut accéder à un fichier partagé, Venus envoie une requête à Vice et celui-ci répond en retournant une copie complète du fichier. Venus garde la copie dans la mémoire secondaire du poste et lui donne un *inode*. Le poste peut donc considérer la copie comme un fichier local. Bien évidemment, Vice se charge du maintien de la cohérence des fichiers cachés.

1.1.2.1 L'espace de désignation de fichiers

Les fichiers partagés sont organisés dans une hiérarchie unique, visible comme telle par tous les postes. Cette hiérarchie répartie, illustrée dans la figure 1.2, est construite à partir d'un ensemble de sous-hiérarchies. Chaque sous-hiérarchie réside dans un seul site serveur et comporte un ou plusieurs points de montage. La racine d'une sous-hiérarchie est montée sur l'un des points de montage d'une autre sous-hiérarchie. En plus, la racine d'une sous-hiérarchie n'est montée que sur *un* point de montage.

La hiérarchie de fichiers de chaque poste est augmentée d'un point de montage, de nom `/cmu`. C'est sur lui qu'est montée la racine de la hiérarchie de fichiers gérée par Vice.

Le parcours du chemin d'accès à un fichier local à un poste est réalisé par les mécanismes habituels d'Unix. Si le fichier est partagé (i.e., si son chemin d'accès comprend la chaîne `/cmu`), le parcours a lieu de la manière suivante : Venus demande à Vice une copie du fichier dont le nom est le premier du chemin d'accès. Si Vice lui rend la copie, Venus la lit en cherchant l'identificateur du fichier dont le nom est le deuxième du chemin. Venus demande ensuite une copie de ce fichier. Si Vice la lui rend, Venus la lit en cherchant l'identificateur du fichier dont le nom est le troisième du chemin. Ces actions se répètent jusqu'à ce qu'on termine le parcours et que l'on obtienne l'identificateur du fichier.

1.1.2.2 La gestion des fichiers partagés

Vice duplique sur tous les sites serveurs quelques-unes des sous-hiérarchies. Il s'agit des fichiers les plus souvent consultés et, si possible, les moins souvent mis à jour. C'est le cas de `/bin`, par exemple.

L'un des sites serveurs se charge de coordonner les autres afin de propager les modifications faites sur un fichier d'une sous-hiérarchie dupliquée. Ce site serveur, qu'on appelle 'le gardien primaire', réalise d'abord les modifications sur la copie du fichier qu'il possède. Il envoie ensuite aux autres sites serveurs, dits les 'gardiens secondaires', la copie modifiée pour que ceux-ci remplacent leurs propres copies.

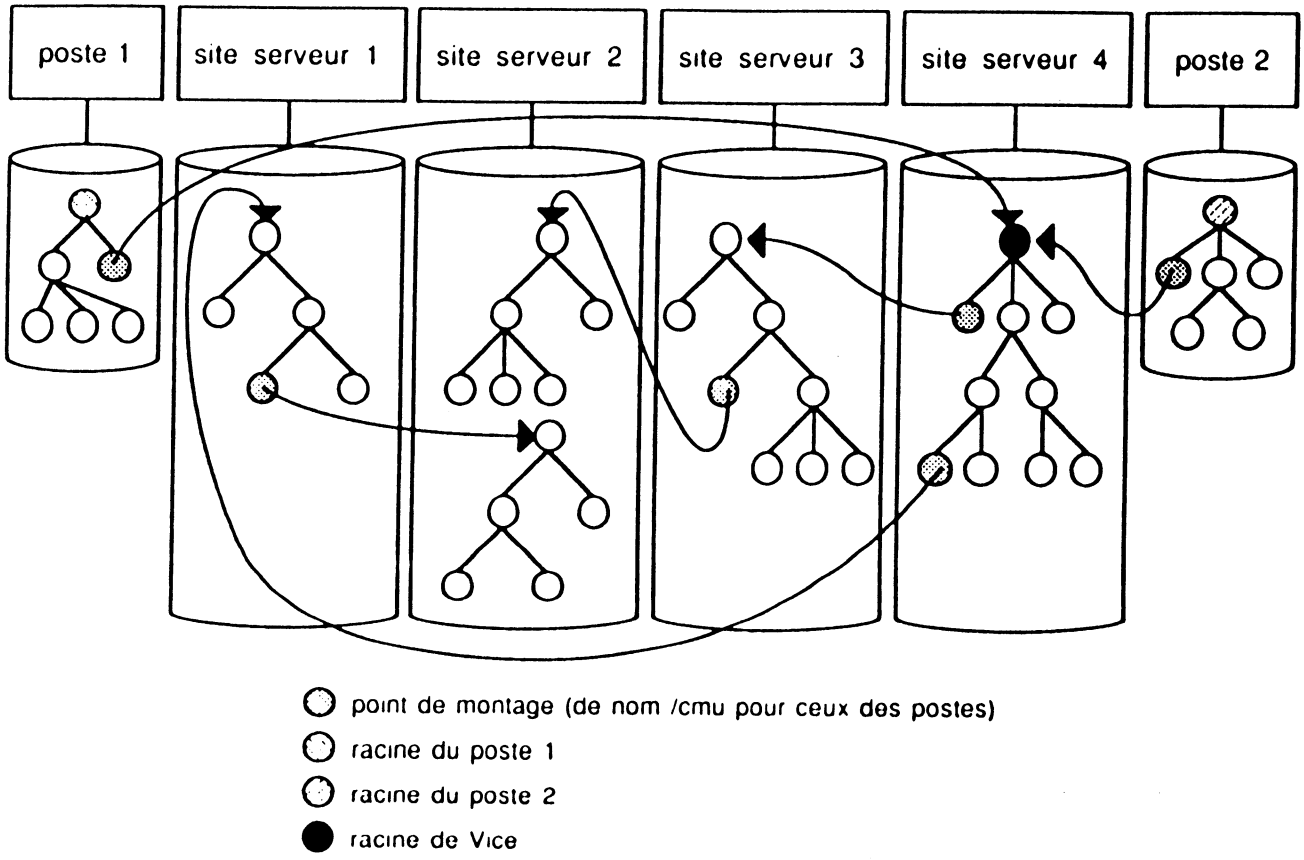


Figure 1.2 - L'espace de désignation de fichiers du système Andrew

Vice réalise le contrôle d'accès à un fichier lorsque le processus Venus d'un poste en demande une copie. Afin de permettre aux applications des postes d'accéder exclusivement aux fichiers partagés, Vice offre la possibilité de les verrouiller et de les libérer. Lorsqu'un poste verrouille un fichier, il est le seul à en posséder une copie. Le verrouillage des fichiers d'une sous-hiérarchie dupliquée est pris en compte par le gardien primaire.

1.1.2.3 L'accès aux fichiers

Le système Unix de chaque poste gère les fichiers locaux et réalise les accès aux copies des fichiers partagés. Les fichiers locaux sont ceux nécessaires pour le *boot* du poste, les fichiers temporaires et les programmes-système qui garantissent le bon fonctionnement du poste lorsque celui-ci est déconnecté du réseau.

Les appels-système du système Unix du poste ont été détournés de façon à établir une communication avec Venus lorsqu'un accès à un fichier partagé est demandé.

Venus envoie une requête à Vice en lui indiquant l'identificateur du fichier. Vice lui répond en envoyant une copie complète du fichier. Venus range la copie dans la mémoire secondaire du poste et lui établit un *inode*. Par ce moyen les applications du poste ne distinguent pas entre les fichiers locaux et les copies des fichiers partagés.

Le transfert dans le sens Venus-Vice ne se fait que si le fichier a été modifié. Vice remplace son fichier par le fichier modifié provenant du poste.

Considérons le cas où deux processus Venus communiquent avec Vice, l'un pour lui demander une copie d'un fichier et l'autre pour lui transférer une copie modifiée de ce même fichier. Si les deux requêtes arrivent en même temps, Vice envoie au premier Venus soit la copie qu'il maintient, soit la copie qui vient d'arriver. Dans les deux cas, il s'agit d'une copie cohérente.

Il se peut que deux (ou plusieurs) postes reçoivent la même copie d'un fichier partagé, qu'ils la modifient et qu'ils la transfèrent à Vice. Etant donnée la manière dont Vice réincorpore les fichiers modifiés, seulement seront visibles dans le système les modifications réalisées sur le poste qui a transféré le fichier en dernier. Le verrouillage de fichiers a été introduit afin de prévenir les incohérences qui peuvent survenir de cette situation.

Une application peut vouloir accéder à un fichier partagée dont une copie est maintenue depuis un certain temps dans la mémoire secondaire de son poste. Or, il est toujours possible que le fichier ait été modifié par une application d'un autre poste. En conséquence, toute copie de fichier partagée doit être validée avant utilisation si l'on veut que sa cohérence soit garantie.

Le processus Venus du poste envoie à Vice une requête de validation. Une réponse positive indique que la copie est bonne et qu'elle peut être utilisée. Une réponse négative oblige Venus à demander une nouvelle copie qui comprenne les modifications du fichier.

A la demande d'une copie d'un fichier, Venus peut solliciter à Vice de lui garder un *callback*. Il s'agit d'un indicateur permettant à Vice d'appeler Venus. Si le fichier est modifié, Vice informe tous les processus Venus ayant demandé un *callback* que leurs copies sont désormais invalides.

Venus autorise l'utilisation des copies avec un *callback* valide sans avoir à lancer au préalable une requête de validation. Pour une copie sans *callback* ou avec un *callback* invalide, Venus doit envoyer la requête.

1.1.2.4 Mise en oeuvre

La communication entre Venus et Vice est de type procédural (appel de procédure à distance sur TCP/IP). Sur chaque site serveur, un ensemble de processus légers (*light-weight process*) se charge de recevoir les requêtes et de les passer au processus-noyau qui implante Vice. Celui-ci peut donc servir de façon concurrente plusieurs requêtes.

Sur chaque site serveur, le processus-noyau qui implante Vice utilise le mode *raw* d'Unix pour la gestion de l'espace des fichiers partagés.

1.2 Oracle : Un système de gestion de bases de données

Oracle est un SGBD centralisé qui utilise un modèle relationnel [Oracle85]. SQL est son langage de définition et de manipulation de données. L'interface *Host Language Call Interface* (HLI) permet aux utilisateurs de communiquer avec Oracle et de lui demander d'exécuter des instructions SQL sur une base de données [Oracle84]. Cette interface est constituée d'un ensemble de primitives qui peuvent être appelées depuis un programme C, Fortran ou Cobol.

Oracle a été développé sur le système Unix V.2.

1.2.1 Organisations physique et logique des données

Une base de données est rangée dans un ensemble de fichiers Unix. Ces fichiers sont constitués de blocs physiques contigus sur le disque. Ils sont par conséquent créés en utilisant le mode *raw* d'Unix. Leur taille, déterminée à la création, est fixe : s'il faut plus de place dans la base de données, un nouveau fichier doit être créé et ajouté.

Une extension est une collection de blocs physiques contigus à l'intérieur d'un fichier.

Du point de vue logique, une base de données comporte un ensemble de partitions, une partition comporte un ensemble de segments, et un segment comporte un ensemble de blocs-Oracle. Une partition s'étale sur plusieurs fichiers. Elle ne grandit qu'en incorporant un nouveau fichier. Un segment est composé d'extensions. Le bloc-Oracle est l'unité logique de stockage. Une liste de blocs-Oracle libres est associée à chaque partition et à chaque segment de la base de données.

Toute relation de la base de données est constituée de deux segments : le segment d'index et le segment de données. Les n-uplets de la relation sont insérés séquentiellement dans les blocs-Oracle du segment de données. Les index de la relation, organisés en arbres-B, sont stockés dans les blocs-Oracle du segment d'index. Quand un bloc-Oracle est plein, on en prend un autre dans la liste de blocs libres du segment. Si cette liste est vide, on continue avec la liste de blocs libres de la partition.

Les mises à jour se font directement sur les données à modifier. Si la donnée modifiée ne tient plus sur le bloc-Oracle, un autre est alors chaîné. Dans un bloc-Oracle, la place libérée par la suppression d'un n-uplet n'est pas utilisée pour l'insertion d'un nouveau. Elle est réservée pour les mises à jour ultérieures des n-uplets qui restent dans le bloc. En conséquence, on ne pourra jamais insérer des n-uplets dans un bloc-Oracle qui a été plein, même s'il y a de la place. Pour ceci, il faut supprimer tous les n-uplets qu'il contient.

Oracle permet le regroupement physique des n-uplets d'une ou plusieurs relations ayant en commun les valeurs d'un ensemble de colonnes (la clé du *cluster*). Il est possible de spécifier la taille du bloc de *cluster*. Dans un tel bloc, la clé du *cluster* est stockée une seule fois.

1.2.2 Contrôle de concurrence

Les transactions sont les unités logiques de travail. Une transaction commence implicitement à la première instruction SQL exécutable rencontrée dans un programme. Elle se termine implicitement à la fin du programme ou explicitement à l'exécution d'une instruction de validation (`commit`) ou d'arrêt (`roll-back`). Si le programme ne finit pas après une telle instruction, une autre transaction commence implicitement.

Le contrôle de concurrence se fait par un mécanisme de verrouillage. La granularité de contrôle est la relation ou le n-uplet. Oracle définit trois types de verrous :

1. Partagé : Autorisé uniquement pour les relations, il permet aux transactions concurrentes de lire les mêmes données mais pas de les modifier.
2. Partagé en mise à jour : Autorisé pour les relations et pour les ensembles de n-uplets d'une relation, il permet aux transactions concurrentes de lire ou de modifier les n-uplets autres que ceux qu'on met à jour.
3. Exclusif : Acquis automatiquement sur les données modifiées, il empêche toute autre transaction d'accéder à ces données.

Une instruction SQL permet aux utilisateurs de choisir parmi ces trois verrous. Une option de SQL peut aussi être utilisée pour verrouiller de manière exclusive l'ensemble de n-uplets qui résulte d'une instruction de sélection.

Lorsque deux transactions entrent en conflit, la plus récente attend la terminaison de l'autre. Les interblocages sont détectés et corrigés : la transaction qui a modifié le moins de blocs-Oracle est arrêtée.

Oracle utilise pour les reprises un journal où sont rangés tous les blocs-Oracle modifiés. En cas de problème avant la fin normale d'une transaction, l'état antérieur de la base est restauré (*rolled back*) en retournant au système les blocs du journal.

1.2.3 La vue de l'utilisateur

Une des primitives HLI établit la communication entre Oracle et l'utilisateur. Les processus qui implantent Oracle et le processus de l'utilisateur communiquent par le biais de la mémoire partagée d'Unix.

Dans le programme de l'utilisateur, les instructions SQL sont codées comme des chaînes de caractères. Pour pouvoir les exécuter, il faut les associer à des 'curseurs'. Un curseur est une zone de mémoire définie dans le programme de l'utilisateur et dont le contenu permet à Oracle de contrôler une instruction SQL.

Avant l'exécution d'une instruction SQL de sélection, l'utilisateur doit indiquer à Oracle l'adresse et la taille de la zone de mémoire où sera placée la valeur de chaque variable de substitution contenue dans l'instruction. S'il s'agit d'une instruction SQL de modification ou d'insertion, l'utilisateur doit associer des valeurs aux variables de susbstitution.

Lors de l'exécution d'une instruction SQL, Oracle réalise la conversion entre sa représentation interne des données et la représentation externe définie dans le programme de l'utilisateur. Après l'exécution d'une instruction SQL de sélection, l'utilisateur peut récupérer, sur les zones définies, les valeurs des variables de substitution de l'instruction. Si l'instruction concerne plusieurs n-uplets, il faut les récupérer un à la fois.

1.3 Etude comparative et conclusions

Les SGBDs mettent en oeuvre des contrôles de concurrence très stricts et se servent des transactions pour garantir la cohérence de leurs données. Dans les systèmes de fichiers, par contre, le partage de données n'est pas aussi fréquent, ce qui permet de réaliser des contrôles moins rigoureux. Cette différence dans la manière de concevoir le partage rend plus ou moins aisée la répartition de la gestion de l'information.

Dans les systèmes de fichiers comme Andrew, le contrôle de concurrence est très souple. Puisque les postes possèdent des copies des fichiers, ils n'ont qu'à les valider pour être en mesure de les utiliser. Même ceci peut être évité par l'emploi des *callbacks*. Le cas où le serveur reçoit en même temps deux copies modifiées d'un fichier est très rare mais les incohérences qui pourraient en résulter sont prévenues par le verrouillage explicite.

Le contrôle qu'exerce NFS est plus sévère que celui d'Andrew car le partage a lieu sur le même site. Cependant, le contrôle se restreint aux informations concernées par une seule requête et n'est pratiqué que pendant la réalistion de l'action demandée dans cette requête. Le serveur ne conserve en outre aucune information qui le lie au client. Les accès exclusifs sont ainsi de courte durée. Des contrôles plus sophistiqués se font à l'aide du service de verrouillage.

En revanche, une base de données est conceptuellement un point focal pour mettre en vigueur le contrôle de concurrence et l'atomicité. C'est précisément le cas d'Oracle qui, par le moyen d'une structure très riche en mémoire secondaire, offre des niveaux de granularité très fins pour l'accès concurrent de plusieurs utilisateurs.

Dans les SGBDs, les structures de données utilisées pour la mise en oeuvre du contrôle de concurrence, de la granularité d'accès et de l'atomicité sont très complexes. D'après Satyanarayanan [Arctic88], leur répartition implique le développement de protocoles très spécialisés et surtout très difficiles à *étendre* (*scale*) sur un grand réseau. L'expérience du système R* [Lindsay84] montre qu'il est possible de répartir un SGBD ; toujours selon Satyanarayanan, il s'agit d'une approche qui 'ne s'étend pas' facilement.

Deux voies ont été suivies jusqu'à présent pour répartir les fonctions des SGBDs. La première consiste à enrichir les systèmes de gestion de fichiers avec des fonctions des bases de données pour le contrôle de la concurrence. Le verrouillage est très évolué dans certains serveurs de fichiers comme XDFS [Sturgis80] [Mitchell82] et Alpine [Brown85]. Une gestion de transactions imbriquées a été mise en oeuvre à l'intérieur du système de gestion de fichiers du système Locus [Popek85].

La deuxième voie consiste à intégrer un service indépendant de bases de données dans un environnement réparti. Cette voie a commencé à être suivie dès qu'il fut évident que le fait de bâtir des bases de données au-dessus des systèmes de fichiers répartis imposait trop de contraintes sur ces systèmes comme, par exemple, la dégradation de la performance et la diminution de la possibilité de s'étendre. Un service de bases de données rend possible 'l'accès réparti', c'est-à-dire, l'accès aux données de la base depuis plusieurs sites et de manière transparente.

Le service d'accès réparti est garanti par un ensemble de serveurs. Lorsque ces serveurs sont indépendants, les clients doivent décomposer leurs requêtes de façon à pouvoir adresser chaque sous-requête à un serveur différent et puis recomposer les résultats obtenus. Un service d'accès réparti de ce type est offert par le système Scylla [Arctic88]. Une autre manière de fournir l'accès réparti est par le biais d'un ensemble *petit* de serveurs fortement couplés. Puisque le nombre de serveurs est réduit, les problèmes de décomposition et optimisation de requêtes, de contrôle de concurrence et d'atomicité peuvent être attaqués efficacement.

Une autre tendance de gestion répartie d'information est celle de fournir des 'morceaux' de mémoire de taille variable à des clients distants tout en contrôlant la concurrence d'accès sur chaque 'morceau' afin de garantir la cohérence de son contenu. A leur tour, les clients mettent en oeuvre les fonctions d'accès aux données qui se trouvent dans ces 'morceaux' de mémoire. C'est la voie que proposent le serveur ObServer et ses clients et qu'on va analyser dans le chapitre suivant.

Le fait que le contrôle de cohérence rende très difficile la répartition d'un SGBD et assez aisée la répartition d'un système de gestion de fichiers *n'implique pas* que la cohérence fournie par les premiers soit *meilleure* que celle des seconds. Même si dans un environment réparti les risques d'incohérence sont plus forts, les systèmes distribués offrent d'aussi bons outils que les centralisés pour garantir la persistance des modifications faites sur leurs données.

Le partage d'objets constitue l'un des aspects fondamentaux du système Guide. Il s'agit d'un système réparti ; cependant, il n'autorise pas la création de copies de ses objets sur plusieurs sites. Un objet est verrouillé de sorte à forcer les environnements d'exécution qui veulent le partager à venir tous dans la mémoire virtuelle du site où il se trouve (cf. chapitre 6). Les accès concurrents sont ensuite synchronisés en utilisant des conditions définies dans sa classe.

On constate que les systèmes NFS et Andrew ont recours à des entités de niveau relativement haut (fichiers de montage) pour cacher les informations de localisation. Il en est de même pour les objets 'représentants' du gestionnaire de mémoire de

Decouchant, qu'on analysera dans le chapitre suivant. Ceci est bien normal, étant donné qu'il s'agit de systèmes qu'on a distribués à partir de systèmes conçus et mis en oeuvre comme des systèmes centralisés. Le cas de Guide est tout autre : dans ce système, la répartition a été prise en compte dès le début. Si elle n'apparaît pas dans les modèles du système, c'est parce qu'on a voulu la cacher aux utilisateurs en faisant d'elle un concept de réalisation et non de modélisation (cf. chapitres 3 et 6).

Dans les SGBDs, la désignation se fait par le biais des prédicats des instructions de sélection. Dans les systèmes de gestion de fichiers, les noms symboliques sont utilisés à cet effet. Il est évident que chaque donnée (n-uplet ou fichier) possède un identificateur interne ; ceci n'est cependant pas rendu visible à l'utilisateur final. Dans les systèmes de fichiers répartis, ces identificateurs internes sont locaux à chaque site. La désignation intersites se fait en utilisant le nom des points de montage. Ceux-ci gardent les informations de localisation, qui restent cachées aux utilisateurs.

Chaque objet du système Guide possède un identificateur unique, indépendant de son site de résidence. En plus de cette désignation interne, un service de désignation symbolique a été mis en oeuvre (cf. chapitre 5). Les noms symboliques sont aussi indépendants des sites de résidence des objets.

2. Tendances actuelles

Dans le domaine de la persistance, les systèmes d'information d'aujourd'hui tendent soit à mettre en oeuvre le modèle de Thatte, soit à fournir les moyens pour la réalisation d'une telle implantation. Dans le deuxième cas, le système se présente comme le serveur d'un autre système, celui qui réalise effectivement le modèle.

En général, les systèmes qui mettent en oeuvre le modèle de la mémoire persistante sont utilisés par d'autres systèmes pour la réalisation d'un modèle de données. C'est le cas des gestionnaires de mémoire du système Smalltalk-80 (cf. section 2.2.1) : l'interprète Smalltalk se sert de cette mémoire pour la mise en place des objets du système.

Un deuxième cas d'utilisation de la mémoire persistante est celui du gestionnaire d'objets OM/WiSS du SGBD O_2 [Vélez89]. Les fonctions d'OM/WiSS sont exécutées au profit des autres modules du SGBD pour la mise en oeuvre d'un modèle à objets [Lécluse89]. En plus d'une mémoire persistante, OM/WiSS facilite les accès associatifs aux objets et fournit un mécanisme de transactions. Ces dernières fonctions sont réalisées en s'appuyant sur la gestion de disque et sur le contrôle de concurrence fournis par le système de stockage WiSS (*Wisconsin Storage System*) [Chou85].

2.1 Le serveur d'objets ObServer

Le serveur ObServer [Skarra86a] [Hornik87] se charge de la gestion de morceaux (*chunks*) de mémoire secondaire que ses clients utilisent pour implanter leurs objets. Les morceaux sont de taille quelconque. Le serveur met aussi en oeuvre un mécanisme non standard de transactions qui permet aux clients de contôler le degré de cohérence de leurs objets.

Deux des clients d'ObServer sont les systèmes Garden et Encore. Garden [Reiss86] est un système de programmation à objets qui s'appuie sur ObServer pour fournir aux utilisateurs un environnement interactif de programmation à objets. Encore [Zdonik86] est un SGBD qui, tout en restant dans la ligne des modèles sémantiques de haut niveau, permet de regarder les données des applications d'un manière très proche de celle des langages à objets.

Les processus qui mettent en oeuvre les clients lient un module d'interface. Celui-ci leur permet d'établir des sessions avec le serveur et d'accéder aux morceaux de mémoire. Une session consiste en une séquence de transactions.

Le serveur supporte plusieurs sessions en même temps. Les clients peuvent se trouver sur la même machine que lui ou sur une machine différente. ObServer lui-même n'est pas distribué.

2.1.1 Mise en oeuvre de la mémoire du serveur

Des clients tels que Garden ou Encore utilisent de nombreux objets au cours d'une transaction. Afin de diminuer les accès à la mémoire secondaire et les transferts entre machines, ObServer met en oeuvre un mécanisme de regroupement d'objets[4] en segments.

Un segment est un ensemble de blocs-disque contigus où se trouve rangé un groupe d'objets conformes à un certain critère. Le maintient de la cohérence oblige à ranger dans un segment les nouveaux objets qui obéissent également au critère. La taille d'un segment est en conséquence variable. Le segment est l'unité d'accès au disque et l'unité de transfert entre machines.

Tout objet appartient à un segment et, en principe, à un seul. Il se peut cependant qu'un objet soit conforme à plus d'un critère. Le serveur offre la possibilité de dupliquer l'objet dans les segments dont il vérifie le critère. Ceci est évidemment pénalisé par le coût des mises à jour. En effet, une modification est répercutée de manière atomique à toutes les copies d'un objet. Un client ne peut donc avoir accès à un objet qui si toutes les copies ont été mises à jour.

Lorsqu'un client demande l'accès à un objet, le serveur lui envoie le segment entier. Bien évidemment, ObServer indique les conditions sous lesquelles le client peut accéder aussi aux autres objets du segment (cf. section 2.1.2.1).

2.1.2 Partage des objets

Pour que les clients puissent partager les objets et maintenir en même temps le niveau de cohérence souhaité, ObServer met en place un schéma particulier de contrôle de concurrence. Les transactions sont mises en oeuvre en utilisant ce contrôle.

2.1.2.1 Contrôle de concurrence

Le serveur met en oeuvre cinq types de verrous : libre, lecture et écriture non exclusives et lecture et écriture exclusives. Le premier type est compatible avec tous les autres. Un verrou en lecture non exclusive permet à un client de lire un objet sans interdire aux autres d'y accéder. Un verrou en lecture exclusive défend aux autres client de modifier l'objet. Un verrou en écriture non exclusive permet aux autres clients l'accès en lecture non exclusive. L'écriture exclusive interdit tout autre accès.

Le client ayant verrouillé un objet[5] peut vouloir être informé du statut de celui-ci à

---

4. On établit désormais une équivalence entre les termes 'objet' et 'morceau de mémoire'.
5. Par abus du langage, on dit qu'un client verrouille un objet. Nous voulons signifier par là que le client demande au serveur de verrouiller l'objet et que celui-ci réalise effectivement cette opération.

la suite des actions des autres clients. En particulier, il voudrait savoir si l'objet a été modifié ou s'il y a d'autres clients le demandant. ObServer met cinq types de notification à la disposition d'un client ayant verrouillé un objet : pas de notification, notification à la suite d'une modification et notification du fait qu'un autre client veut lire ou modifier, seulement lire ou seulement modifier l'objet.

Etant donné qu'il reçoit un segment entier comme reponse à sa requête d'accès à un objet, un client doit déterminer ce qu'il va faire des autres éléments du segment. Puisque, selon le critère de groupement, les autres objets ont une grande probabilité d'être lus ou modifiés, le client doit pouvoir les verrouiller. Ainsi, en plus du verrou qu'il demande sur l'objet auquel il veut accéder, ObServer permet au client de définir un autre verrou dont la portée est le reste du segment.

2.1.2.2 Transactions

Une transaction est une suite atomique d'interactions entre un client et le serveur. Cette suite possède les caractéristiques suivantes :

- Elle peut être de longue durée.
- Elle est bien délimitée : interactions de commencement, validation (`commit`) et arrêt (`abort`).
- L'une des interactions qui la composent peut, à son tour, être une transaction. Il y a donc de l'imbrication de transactions.
- Les objets verrouillés au cours de la transaction sont libérés à la fin de celle-ci. Une option permet cependant de libérer un objet à un moment quelconque de la transaction.
- Les modifications réalisées au cours de la transaction sont validées à la fin de celle-ci. Une option permet cependant de valider les modifications réalisées sur un objet à un moment quelconque de la transaction. Toute modification validée est visible après la validation et ne peut pas être défaite.
- Lorsqu'une transaction est arrêtée, seules sont défaites les modifications non encore validées.

L'ensemble de ces caractéristiques permet aux clients de définir les transactions qui s'ajustent le plus à leurs propres besoins. Il y a, en effet, des applications dont le besoin de cohérence n'est pas aussi fort que celui garanti par la sérialisation de transactions. Bien évidemment, la sélection des verrous appropriés et le bon choix des moments de verrouillage, de libération et de validation permettent la mise en oeuvre des transactions basées sur le 'protocole à deux phases' [Balter85] [Delobel82].

2.1.2.3 L'accès aux objets

Les demandes des clients concernent, d'un part, les accès aux objets et, de l'autre, les verrouillages et les libérations des objets et les validations des modifications. Pour accéder à un objet, le client indique son identificateur et le verrouillage (le verrou et le mode de notification) qu'il veut sur l'objet et sur les autres éléments du segment.

Le serveur satisfait la demande en répondant aux trois questions suivantes :

1. Est-ce que le client a déjà l'objet qu'il demande ?
2. Si oui, est-ce que cet objet comporte déjà les modifications les plus récentes ?
3. Si la réponse aux questions antérieures est non, quel segment envoyer ?

Pour répondre à la première question, ObServer maintient une liste dont chaque élément associe l'identificateur d'un client aux identificateurs des segments qui lui ont été envoyés.

Si le client n'a pas l'objet et si le verrou peut être garanti, le serveur envoie au client le segment entier où se trouve rangé l'objet. Ceci répond à la troisième question. Le module lié à chaque client empêche que celui-ci accède aux objets du segment verrouillés par les autres clients. Si le client a déjà le segment et si le verrou peut être garanti, le serveur n'envoie que l'autorisation d'accès.

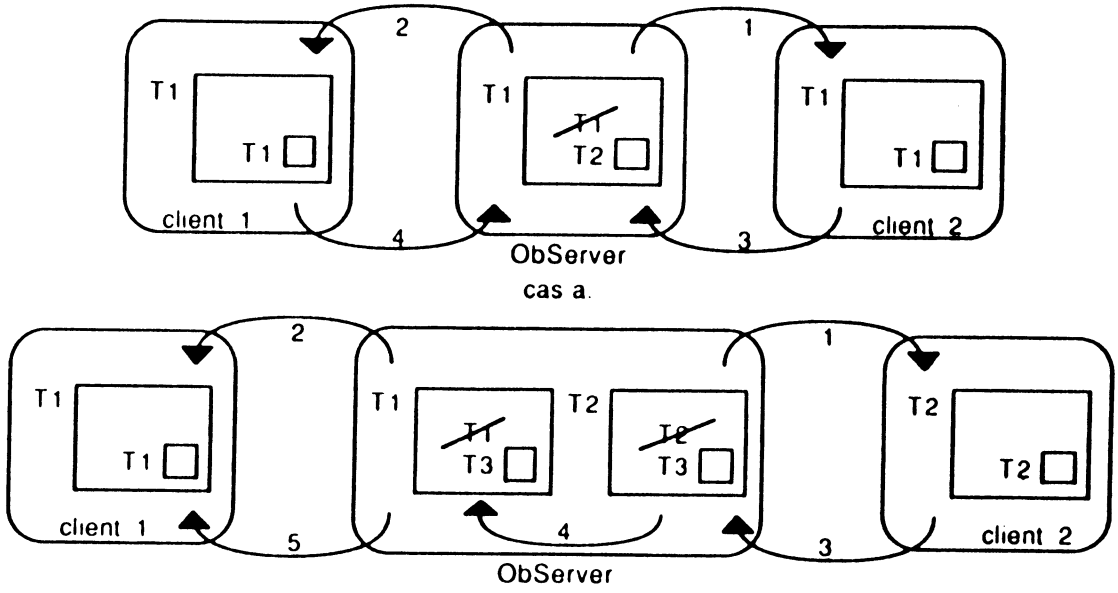
Le serveur a pu envoyer le même segment à plusieurs clients. La cohérence des objets dans les copies des segments est garantie par le biais des verrous. A tout moment, la copie qui se trouve dans le serveur comporte les dernières modifications validées faites sur les objets du segment.

ObServer utilise des estampilles pour suivre les traces des modifications des objets. Avant de le transférer pour la première fois, le serveur affecte la même estampille au segment et à tous ses objets. A la suite d'une modification sur un objet, une nouvelle estampille est affectée à cet objet. Pour répondre à la deuxième question, le serveur compare l'estampille de l'objet et celle du segment. Si les valeurs sont différentes et si le verrou peut être garanti, l'objet est envoyé au client.

La figure 2.1 montre l'utilisation des estampilles pour l'envoi d'objets. La figure illustre également le cas où deux segments différents mais comportant des copies du même objet sont envoyés à deux clients.

A l'arrivée d'un nouveau segment, le module lié au client ajoute une entrée à sa table des segments en mémoire principale. Une entrée de cette table lui permet de gérer les objets d'un segment. En particulier, une entrée maintient les informations de gestion du débordement nécessaires pour pouvoir élargir les objets.

Un client peut demander au serveur de créer un objet. Celui-ci lui alloue un identificateur mais ne lui réserve pas d'espace. Par la suite, ObServer et ses clients voient la création des blocs de l'objet comme l'élargissement d'un objet de taille 0. Pour la destruction d'un objet, le client envoie au serveur l'identificateur.



cas a.

0. Le segment et l'objet ont la même estampille T1
1. Le client 2 reçoit le segment afin de modifier l'objet
2. Le client 1 reçoit le segment mais ne peut pas, pour l'instant, modifier l'objet
3. Après l'avoir modifié, le client 2 rend l'objet au serveur; le serveur lui donne une nouvelle estampille
4. Lorsque le client 1 demande l'accès à l'objet, le serveur doit lui envoyer une nouvelle copie

cas b.

0. Chaque segment et son objet ont la même estampille : T1ou T2.Il s'agit en fait de deux copies du même objet
1. Le client 2 reçoit le segment ayant l'estampille T2 afin de modifier la copie de l'objet
2. Le client 1 reçoit le segment ayant l'estampille T1 afin de modifier la copie de l'objet
3. Après l'avoir modifié, le client 2 rend l'objet au serveur; le serveur lui donne une nouvelle estampille
4. Les copies son modifiées de façon atomique (il en est de même pour les estampilles)
5. Le serveur indique au client 1 que sa copie n'est plus valide.

Figure 2.1 - Utilisation des estampilles

Le module d'interface d'un client n'envoie au serveur que les blocs modifiés et les blocs de débordement des objets d'un segment. ObServer utilise aussi une table de segments pour la gestion de sa mémoire principale. A l'arrivée des blocs envoyés par un client, il les installe dans ses structures en mémoire principale avant de les ranger en mémoire secondaire. Si un segment comporte des blocs de débordement, tous ses blocs sont copiés de façon contiguë dans une nouvelle zone de mémoire principale allouée à

cet effet. Ainsi, une seule opération d'entrée/sortie suffit pour l'écriture sur disque d'un segment modifié.

2.1.3 Les clients

ObServer ne manipule que des morceaux de mémoire. La mémoire persistante est mise en oeuvre par les clients. Ceux-ci se chargent aussi de gérer les objets comme des instances de types.

Dans le SGBD Encore, un type est une spécification de comportement. Il décrit un ensemble d'opérations, un ensemble de propriétés et un ensemble de contraintes. Une opération d'un objet est une instance d'un type particulier utilisée pour accéder à l'objet. Les propriétés, également des instances d'un type particulier, permettent d'établir des rapports entre objets. Les propriétés peuvent être mono- ou multi-valuées. Deux propriétés particulières sont `is-a`, qui induit une relation d'héritage entre les types, et `part-of`, qui exprime la composition d'objets. Toutes les instances d'un type *T* doivent satisfaire les contraintes définies dans *T*.

De manière similaire à Encore, le système Garden définit un schéma particulier de types. Ce schéma lui permet de supporter une grande variété de langages graphiques. Les objets sont utilisés pour décrire les images structurées, qui sont les outils de base de la programmation graphique. En plus des objets fournis par ObServer, Garden implante des objets qui lui sont locaux. Afin de traiter de manière homogène ses objets et ceux d'ObServer, ce client définit ses propres verrous, transactions et schéma de récupération d'espace. Pour la mise en oeuvre de ces outils, Garden utilise, lorsqu'il est nécessaire, les outils correspondants fournis par ObServer.

2.1.4 Mise en ouvre

ObServer et les modules d'interface de ses clients ont été réalisés au-dessus du système Unix 4.2 BSD. Etant donné qu'Unix ne fournit pas un stockage contigu pour ses fichiers, la mise en oeuvre des segments utilise le mode *raw*. Les communications sont implantées en utilisant les *sockets* d'Unix en mode *stream* (circuit virtuel).

2.2 Mémoires persistantes

On appelle **mémoire persistante** toute réalisation du modèle de Thatte. Le fait de lier la persistance au ramasse-miettes est mis en oeuvre par bien des systèmes. Dans le domaine des SGBDs à objets, en plus d'Encore, on peut en citer trois autres, à savoir, O_2, Orion [Kim88] et GemStone™ [Maier87]. Dans cette section, nous nous limitons à

™ GemStone est une marque déposée de *Servio Logic Development Corp.*

présenter trois réalisations du modèle (dont la propre implantation de Thatte) en laissant de côté tous les aspects inhérents aux modèles de données.

2.2.1 Une mémoire persistante pour Smalltalk-80

Tous les objets du système Smalltalk-80 standard se trouvent en mémoire principale : la gestion d'objets fait partie intégrante de la couche d'interprétation [Goldberg83]. Dès qu'on veux que le nombre d'objets ne soit pas limité à la capacité de la mémoire principale, se pose le problème de la réalisation d'une mémoire d'objets à l'aide de deux niveaux de mémoire physique.

Afin de mettre le plus facilement en évidence les problèmes des systèmes à objets, il convient de gérer les objets en tant que tels. Un exemple de ces problèmes est le défaut d'objet : Le gestionnaire de la mémoire d'objets doit pouvoir savoir si un objet déterminé est présent en mémoire principale. Si ce n'est pas le cas, il doit, pour l'y amener, disposer de la place nécessaire ou être en position de libérer de la place pour le loger.

Ces problèmes sont inhérents à toute mémoire segmentée ou paginée. L'aspect nouveau de la mémoire d'objets réside dans ses caractéristiques quantitatives. Dans les systèmes segmentés classiques, les segments sont en petit nombre et de grandes tailles ; dans la mémoire du système Smalltalk-80, les objets sont très nombreux et de petite taille. En plus, cette mémoire est persistante.

La réalisation d'une mémoire à deux niveaux pose un problème à l'interprète Smalltalk-80 : dans le cas où un défaut d'objet arrive lors du traitement d'un octet-code, l'interprète doit interrompre le déroulement du processus Smalltalk courant. Ceci doit s'accompagner de la sauvegarde des registres qui contiennent l'état d'avancement pour le traitement de l'octet-code. En somme, un défaut d'objet a pour conséquence de faire perdre à l'interprète son caractère séquentiel. C'est un problème auquel s'attaquent les deux réalisations de la mémoire persistante que l'on analyse par la suite.

2.2.1.1 LOOM : Une réalisation centralisée de la mémoire d'objets

LOOM (*Large Object-Oriented Memory*) est un système de mémoire virtuelle mono-utilisateur qui gère et échange les objets entre deux niveaux de mémoire physique [Kaehler86]. De même que OOZE, le système de mémoire virtuelle développé pour Smalltalk-76 [Kaehler81], LOOM gère et transfère des objets.

Les réalisateurs de LOOM ont voulu éviter au maximum les modifications à l'interprète Smalltalk. Ils ont donc conservé le même espace d'adressage pour les objets en mémoire principale : pointeurs à 16 bits. En conséquence, l'interprète peut accéder aux objets en mémoire principale de la même manière que dans le système Smalltalk à mémoire résidante.

Afin de pouvoir disposer d'un grand nombre d'objets, un espace d'adressage basé sur des pointeurs à 32 bits a été mis en oeuvre en mémoire secondaire. Une traduction d'adresses s'avère donc nécessaire chaque fois qu'un objet est échangé entre les deux niveaux de mémoire. D'une part, tout objet en mémoire principale conserve, à une position prédéfinie de son état, son adresse en mémoire sécondaire. De l'autre, une fonction de dispersion permet d'obtenir son adresse en mémoire principale à partir du pointeur en mémoire secondaire.

La figure 2.2 illustre le cas d'un objet O1 qui maintient un pointeur sur l'objet O2. Tant que O1 se trouve en mémoire sécondaire, le pointeur sur O2 sera un pointeur en mémoire secondaire. Lorsque l'interprète veut accéder à O1, celui-ci est chargé et le pointeur sur O2 est changé pour le pointeur en mémoire principale qui résulte de l'application de la fonction de dispersion. Si O2 est chargé, O1 pointe désormais sur lui. Si O2 n'est pas encore chargé, O1 pointe sur une 'feuille' qui représente O2 en mémoire principale.

La feuille comporte le pointeur en mémoire secondaire de O2. Ce pointeur sera utilisé pour charger O2 ultérieurement. Au moment de son chargement, O2 remplace la feuille qui le représente. Lorsqu'il est rangé en mémoire secondaire, la feuille occupe à nouveau sa place. La feuille n'est détruite que lorsque O1 lui-même est rangé en mémoire secondaire. Plus précisément, la feuille d'un objet O reste en mémoire principale tant qu'un objet comportant un pointeur sur O reste aussi en mémoire principale.

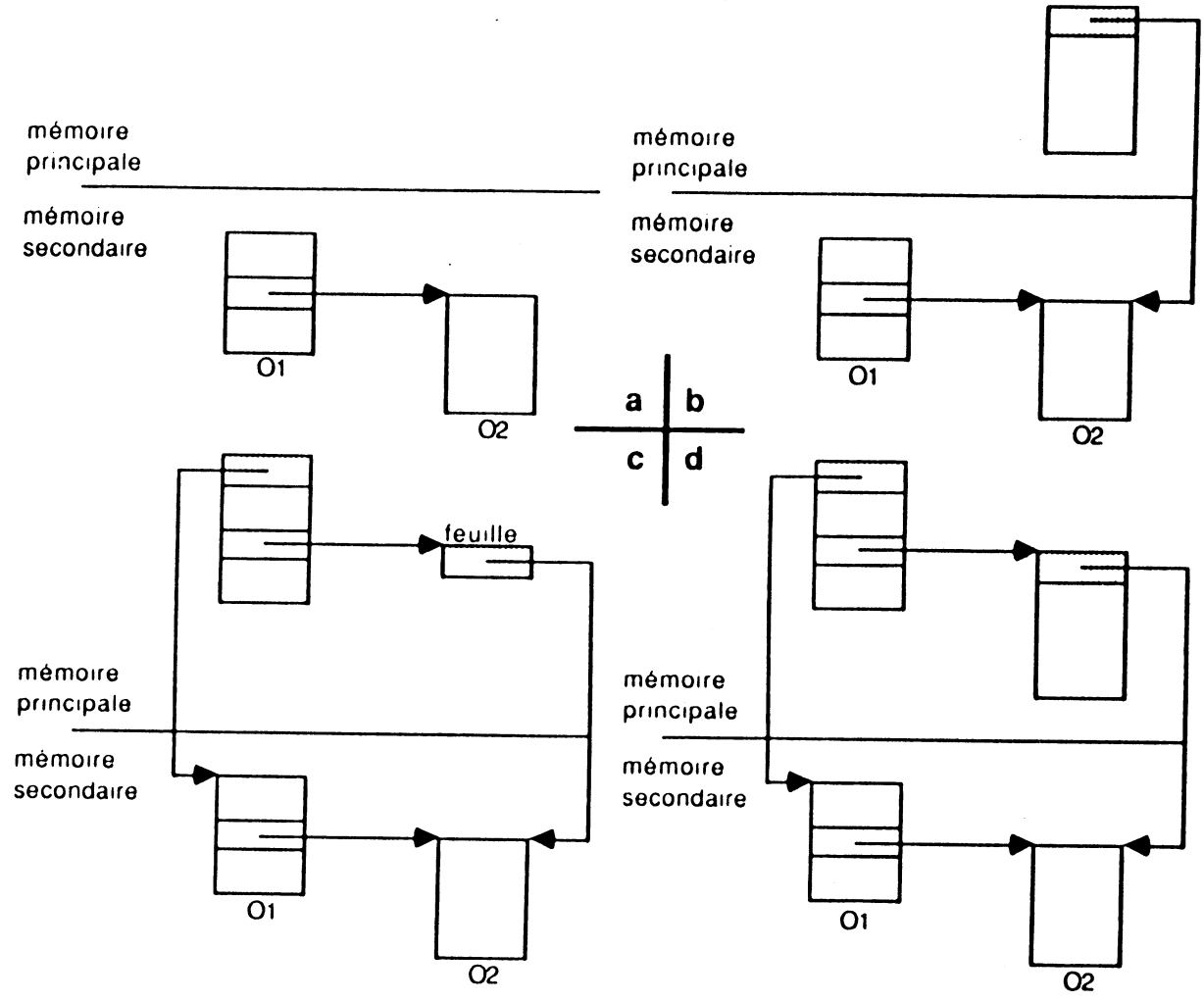
Les objets qu'on vient de créer et qui n'ont pas encore été rangés en mémoire secondaire sont dits 'non mûrs'. L'objet devient 'mûr', et acquiert une adresse en mémoire secondaire, quand un objet qui comporte un pointeur vers lui est rangé en mémoire secondaire. Un problème peut se poser pour ces objets qui 'mûrissent' : étant donné qu'ils possèdent déjà un pointeur en mémoire principale, à l'allocation d'un pointeur en mémoire secondaire, la fonction de dispersion peut ne pas marcher correctement.

Le problème est résolu par l'introduction des 'blocs d'indirection'. Le résultat de l'application de la fonction de dispersion est un pointeur en mémoire principale à un bloc d'indirection. Ce bloc comporte le pointeur en mémoire principale sur l'objet qui vient de 'mûrir'.

Lorsqu'il détecte un défaut d'objet, l'interprète Smalltalk sauvegarde ses registres et passe le contrôle à LOOM. Il reprend ensuite le contrôle. Dans certains cas, l'interprète doit conserver un état spécial et et reprendre le contrôle à partir de cet état. Les détails se trouvent dans [Kaehler86].

Le ramassage de miettes

Un objet possède deux compteurs : le nombre d'objets qui pointent sur lui en mémoire principale et le nombre d'objets qui pointent sur lui en mémoire secondaire. N'ayant pas de représentation en mémoire sécondaire, les objets 'non mûrs' n'utilisent



1. O2 n'est pas chargé :
   avant le chargement d'O1,
   c'est le cas a
2. O2 n'est pas chargé :
   on charge O1, c'est le cas c.
3. O1 et chargé, on charge O2
   c'est le cas d

2'. O2 est chargé :
   avant le chargement d'O1,
   c'est le cas b.
3'. O2 est chargé :
   on charge O1, c'est le cas d.

4. O1 est chargé :
   on décharge O2, c'est le cas c.
5. O2 n'est pas chargé :
   on décharge O1, c'est le cas a.

Figure 2.2 - Objets et feuilles

que le premier compteur.

Lorsque le premier compteur d'un objet est à 0, il peut être éliminé de la mémoire principale. Si le deuxième est aussi à 0, il est détruit complètement.

2.2.1.2 La réalisation répartie de Decouchant

L'inconvénient de LOOM réside dans l'inévitable traduction entre les deux espaces d'adressage. En choisissant de représenter de la même façon les objets dans les deux niveaux de mémoire, Decouchant met en oeuvre un seul espace d'adressage dans son gestionnaire d'objets[6] [Decouchant87]. Ce gestionnaire est par ailleurs distribué, ce qui présente comme avantages la communication entre utilisateurs, le partage d'objets et la répartition de la charge. De même que dans LOOM, la couche interprète et le gestionnaire sont indépendants.

La désignation est transparente à la localisation des objets sur le réseau. Afin de réaliser cette transparence de localisation physique, deux considérations ont été prises en compte :

1. Si un objet O détient une référence, cette référence identifie *toujours* un objet résidant sur le même site que O. Il n'existe donc pas d'espace global d'adressage.
2. Tout objet distant possède un représentant local. Il s'agit d'un objet de type particulier et invisible à l'utilisateur. A tout instant, cependant, un objet peut venir prendre la place de son répresentant.

La figure 2.3 illustre ces deux considérations. Afin de désigner un objet distant, les objets du site x utilisent des références vers son seul répresentant local sur x.

Le gestionnaire d'objets est réalisé comme un ensemble de gestionnaires locaux coopérants.

2.2.1.2.1 Structure d'un gestionnaire local d'objets

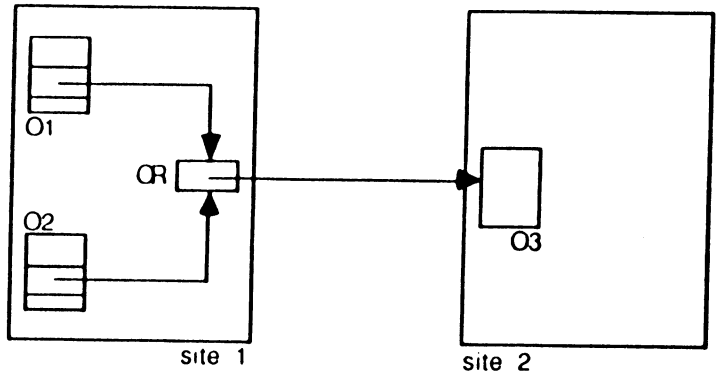
La gestion des objets locaux est réalisée par les trois modules suivants : Le gestionnaire de la mémoire principale (GMP), le gestionnaire de la mémoire secondaire (GMS) et le gestionnaire du réseau (GR).

Le GMP résout les défauts d'objet locaux au site et gère l'espace libre de la mémoire principale. Il utilise l'interface que lui offre le GMS pour charger et décharger les objets depuis et vers la mémoire sécondaire. Un seul processus exécute la gestion de la mémoire principale (le PGMP).

Le GMS se charge de la gestion d'objets en mémoire secondaire. C'est lui qui détermine l'identificateur de tout nouvel objet crée. Il s'agit d'un identificateur unique qui se trouve dans toutes les références vers l'objet. Un seul processus exécute la gestion de la mémoire principale (le PGMP).

---

6. Decouchant utilise le terme 'gestionnaire d'objets' à la place du terme 'gestionnaire de la mémoire d'objets'.



Les réferences de O1 et O2 sont toujours locales au site 1
OR est le représentant de O3 sur le site 1

Figure 2.3 - Objet représentant

Le GR traite les demandes d'accès à des objets distants et garantit la correction des migrations et du ramasse-miettes réparti. Il assure la communication avec les autres GRs du réseau. Un seul processus exécute la gestion du réseau (le PGR). Il est est le processus maître du système.

2.2.1.2.2 Les deux niveaux de mémoire

Decouchant ne considère plus un interprète Smalltalk unique mais autant d'interprètes que de processus à interpréter. Il existe donc, dans chaque site du réseau, un noyau de synchronisation de processus qui gère l'exécution (création, destruction, suspension, synchronisation) des divers processus-interprètes.

Pour accéder aux objets, un processus-interprète utilise les procédures standards Smalltalk (lecture ou écriture du descripteur ou de l'état d'un objet, création d'un objet, etc.). Afin de faire face à un défaut d'objet, chaque procédure standard est 'parenthésée' de deux procédures spéciales. La première garantit la présence de l'objet en mémoire principale et le verrouille selon le type d'accès demandé. Elle bloque le processus-interprète si l'accès n'est pas autorisé. La deuxième fonction libère l'objet. En fait, les procédures 'de parenthèse' constituent l'interface que le GMP offre aux processus-interprètes.

Un processus-interprète informe le PGMP qu'il a détecté un défaut d'objet et se met en attente. Le PGMP détermine si l'objet est effectivement absent de la mémoire : il pouvait être en cours de chargement lorsque le processus-interprète a détecté le défaut. Dans ce cas, le processus-interprète est réveillé. Si l'objet n'est pas présent, le PGMP demande le chargement au PGMS. Celui-ci charge l'objet et répond au PGMP. Les processus-interprètes en attente pour l'objet sont ensuite réveillés.

Un processus-interprète informe le PGMS qu'il veut créer un objet et se met en attente. Le PGMS crée un nouvel identificateur d'objet, réserve l'espace nécessaire en mémoire secondaire pour l'objet et réveille le processus-interprète. Tout se passe ensuite comme si l'objet existait déjà et comme s'il n'était pas présent en mémoire principale.

2.2.1.2.3 La mise en oeuvre de la répartition

Un représentant d'un objet O comporte l'identificateur du site de résidence de O, disons *x*, et une référence vers O locale à *x*. Ce mécanisme permet donc la réalisation des accès distants.

Un processus-interprète informe le PGR qu'il accède au représentant d'un objet et se met en attente. Le PGR communique avec son homologue du site possédant l'objet réel. Celui-ci réalise sur son site l'accès souhaitér et et retourne le résultat au PGR du site émetteur. Ce PGR réveille le processus-interprète qui reçoit le résultat de l'accès et continue son exécution.

Lorsqu'un objet migre, un représentant doit lui être créé sur son site original de résidence. Egalement, un représentant est créé lorsqu'on veut accéder à un objet distant qui n'a pas de représentant local.

Sur un site *x*, un objet distant ne comporte qu'un seul représentant. Cet objet disparait dès que l'objet revient sur *x*. Toute migration d'objet sur un site *x* détermine donc d'abord si sur *x* existe un représentant.

Quand un objet change de site, les informations maintenues par tous ses représentants doivent être mises à jour. Or, cette mise à jour n'est pas répercutée immédiatement sur le réseau. Elle se fera progressivement, lors des tentatives d'accès à l'objet par le biais des représentants non à jour. Ceci est illustré par la figure 2.4.

Supposons que, sur le site *x*, il existe un représentant d'un objet qui se trouve réellement sur le site *y* (cas a. de la figure). Si l'objet migre du site *y* au site *z*, le représentant sur *x* n'est plus à jour. Cependant, les informations qu'il comporte lui permettent de désigner le représentant que l'objet ayant migré laisse sur *y* (cas b.). Si sur *x* on veut accéder à l'objet, on va d'abord le chercher sur *y*. Le représentant sur *y* indique que l'objet se trouve désormais sur *z*. C'est alors que le représentant sur *x* est mis à jour de sorte à désigner le site *z*.

2.2.1.2.4 La récupération de l'espace des miettes

Tout objet comporte un compteur de références externes, égale au nombre de ses représentants sur le réseau. Il comporte aussi le nombre d'objets qui le référencent sur son site de résidence. Si le compteur de références locales tombe à 0 mais l'objet continue à avoir de représentants, il doit migrer sur l'un des sites où se trouve un représentant.

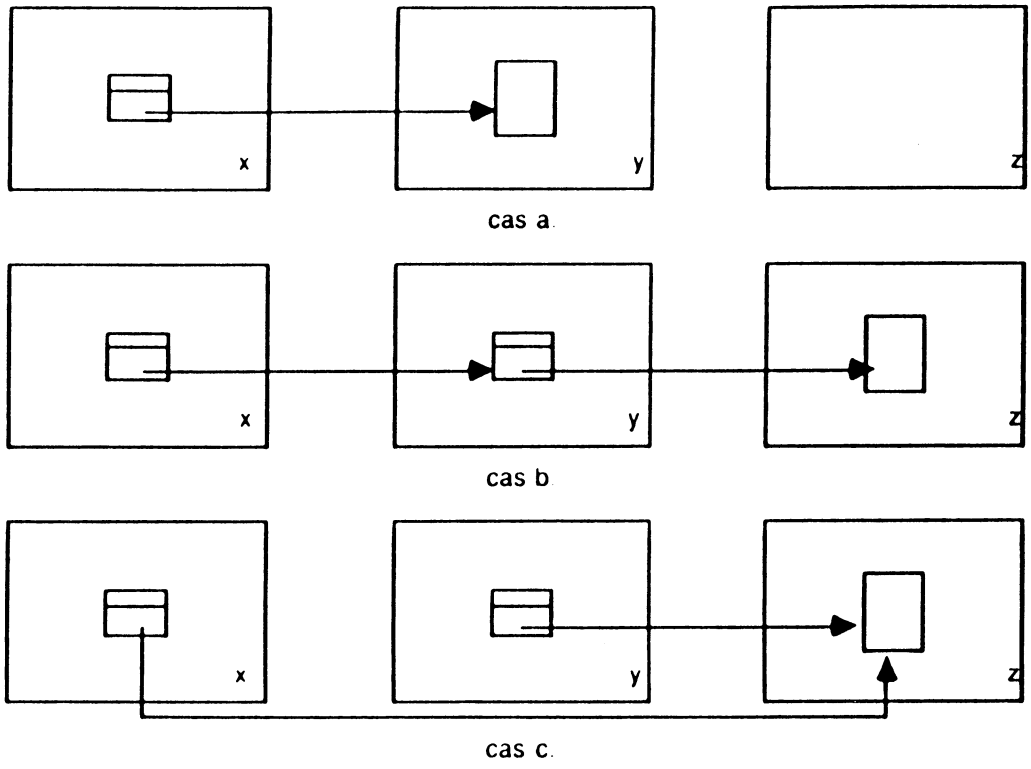


Figure 2.4 - Mise à jour d'un représentant

Un objet doit être détruit lorsque son compteur de références locales est 0 et qu'il n'a plus de représentant. Un processus-interprète informe le PGMP qu'il a détecté cette situation. Le PGMP détruit l'objet en mémoire principale et demande au PGMS de le détruire en mémoire secondaire.

Tous les sites doivent se concerter pour mettre en oeuvre le marquage qui permettra de déceler les cycles répartis de miettes. Les processus-interprètes de tous les sites sont suspendus. Une marque est émise depuis la racine persistante de chaque site. Les cycles de miettes locaux à chaque site n'étant pas marqués, ils peuvent ensuite être éliminés.

Lorsque le parcours de la hiérarchie d'objets arrive à un représentant, le site émet un message de propagation de marque vers le site où se trouve l'objet représenté. Si cet objet n'est pas encore marqué, il devient une nouvelle racine pour son site et le processus de marquage se propage à partir de lui.

A un moment donné, un site considère qu'il a fini le marquage local et celui provoqué par les messages de propagation qu'il a reçus jusqu'a présent. Il envoie alors à tous les sites du réseau un message de terminaison de marquage. Cet algorithme réparti s'arrête lorsque *chaque* site a reçu de *chacun* des autres sites le message de terminaison.

A tout moment, un site peut recevoir d'un site *quelconque* un message de propagation. Si un tel message arrive sur un site après que celui-ci ait émis des

messages de terminaison, alors tous les autres sites invalident ces messages et se mettent à nouveau en attente d'un message de terminaison.

2.2.1.3 Mise en oeuvre de la mémoire persistante de Smalltalk-80

LOOM a été implantée sur l'ordinateur *Xerox Dorado*, une machine à 16 bits et dotée de 4K octets de microcode. Ce microcode comporte l'interprète Smalltalk-80, modifié légèrement pour accueillir LOOM. Ces modifications concernent le code nécessaire pour passer le contrôle à LOOM lors des défauts d'objet et des autres opérations de gestion de la mémoire. LOOM lui-même a été réalisé en code machine.

Decouchant a mis en oeuvre sa mémoire d'objets au-dessus d'Unix 4.2 BSD sur un Sun 2/50. Un multiplexeur d'activités au sein d'un processus Unix lui permet de réaliser les processus-interprètes, le PGMP et le PGMS. Un deuxième processus Unix réalise le PGR. Le multiplexeur gère les entrées/sorties séparément et les rédistribue sur chaque ctivité. Les processus communiquent par des *sockets*.

2.2.2 La mémoire persistante de Thatte

La réalisation de Thatte d'une mémoire persistante [Thatte86] cherche à maintenir la cohérence des objets vis a vis des pannes du système ou des disques. Thatte insiste sur le fait que l'implantation directe de cette mémoire sur une machine dont le matériel est libre de défaillances ne présente pas d'avantages. En effet, des erreurs de logiciel peuvent toujours faire tomber le système en panne et ces erreurs sont bien plus fréquentes que les erreurs matérielles.

En ce qui concerne la réalisation d'un mécanisme de maintien de cohérence face aux pannes, les ordinateurs symboliques, tels que celui où Thatte réalise sa mémoire (un *TI Explorer™ Lisp Machine*), présentent le problème de la structure des objets : un objet peut tenir dans une page virtuelle seulement ou en occuper plusieurs, plusieurs objets peuvent se trouver dans la même page, les pointeurs peuvent traverser les limites d'une page, etc.

Ainsi, si le rangement d'un objet en mémoire sécondaire peut garantir, à un moment donné, la cohérence de cet objet, il ne peut nécessairement pas garantir celle des rapports qu'il a avec d'autres objets. Pour ceci, il faudrait que tout l'état du système soit écrit sur disque. L'implantation de Thatte vise précisément à maintenir cette cohérence, en réduisant au maximum le temps de sauvegarde sur disque.

™ Explorer est une marque déposée de *Texas Instruments Incorporated.*

2.2.2.1 Le support à la mémoire persistante

La mémoire persistante est mise en oeuvre au-dessus de la mémoire virtuelle paginée de la *TI Explorer Lisp Machine*. Cette mémoire virtuelle comporte un ramasse-miettes qui collecte, de manière automatique, les pages virtuelles qui contiennent des miettes. Pour la réalisation, Thatte a, d'une part, enrichi la mémoire virtuelle des fonctions de maintien de la cohérence face aux pannes du système et, de l'autre, modifié le ramasse-miettes de façon à l'adapter à ces fonctions.

2.2.2.1.1 Résistance aux pannes

Le mécanisme de maintien de cohérence utilise les mêmes idées que celui de quelques SGBDs conventionnels : l'état valide des informations n'est pas touché avant une nouvelle validation ; c'est une copie de cet état qui est consultée et subit des modifications. La caractéristique propre à l'implantation de Thatte réside dans le fait que la granularité du mécanisme est la page virtuelle de mémoire et que les validations sont des points de contrôle communs à toutes les pages et pendant lesquels toutes les pages 'sales' sont écrites sur disque. Après une panne, le système est ramené au dernier état valide. Le mécanisme est indépendant des applications et transparent à l'utilisateur.

Une page virtuelle qui risque d'être modifiée est matérialisée en mémoire secondaire par une paire de blocs-disque non nécessairement contigus. Une page virtuelle qui ne subit presque pas de modification n'est matérialisée que par un bloc-disque. Toutefois le système offre la possibilité de passer d'une réprésentation à l'autre : d'un seul bloc à une paire, dans le cas d'une modification de la page virtuelle correspondante au bloc initial, ou d'une paire à un bloc seul, dans le cas de la paire de blocs correspondantes à une page virtuelle qui n'a pas été modifiée depuis un certain temps.

Dans une position prédéfinie de la mémoire virtuelle se trouve, maintenue de manière fiable, la date du dernier point de contrôle (de validation). Les blocs-disque sont identifiés par des estampilles allouées au moment de l'écriture sur disque de la page correspondante. Les blocs dont l'estampille est plus petite que la date du point de contrôle appartiennent au dernier état valide. Ils ne sont pas modifiés car c'est à partir d'eux que le système redémarre à la suite d'une panne.

Considérons le cas d'un bloc qui, à la suite d'un défaut de page, a été ramené en mémoire principale. Ceci est illustré dans la figure 2.5, cas a. Si la page est modifiée (ce qui est, en principe, très rare), le bloc n'est pas réécrit mais il est transformé en paire : un bloc maintient l'ancien état valide et l'autre comporte l'état actuel modifié.

Dans le cas d'une paire, le bloc qui doit être ramené en mémoire virtuelle, à la suite d'un défaut de page, est celui qui comporte l'estampille la plus grande. Ce bloc contient soit l'état valide de la page, si celle-ci n'a pas subi de modification depuis le dernier point de contrôle, soit l'état modifié de la page, si celle-ci a été modifiée depuis le dernier point de contrôle. Ceci est illustré par les cas b. et c. de la figure 2.5. Si la page

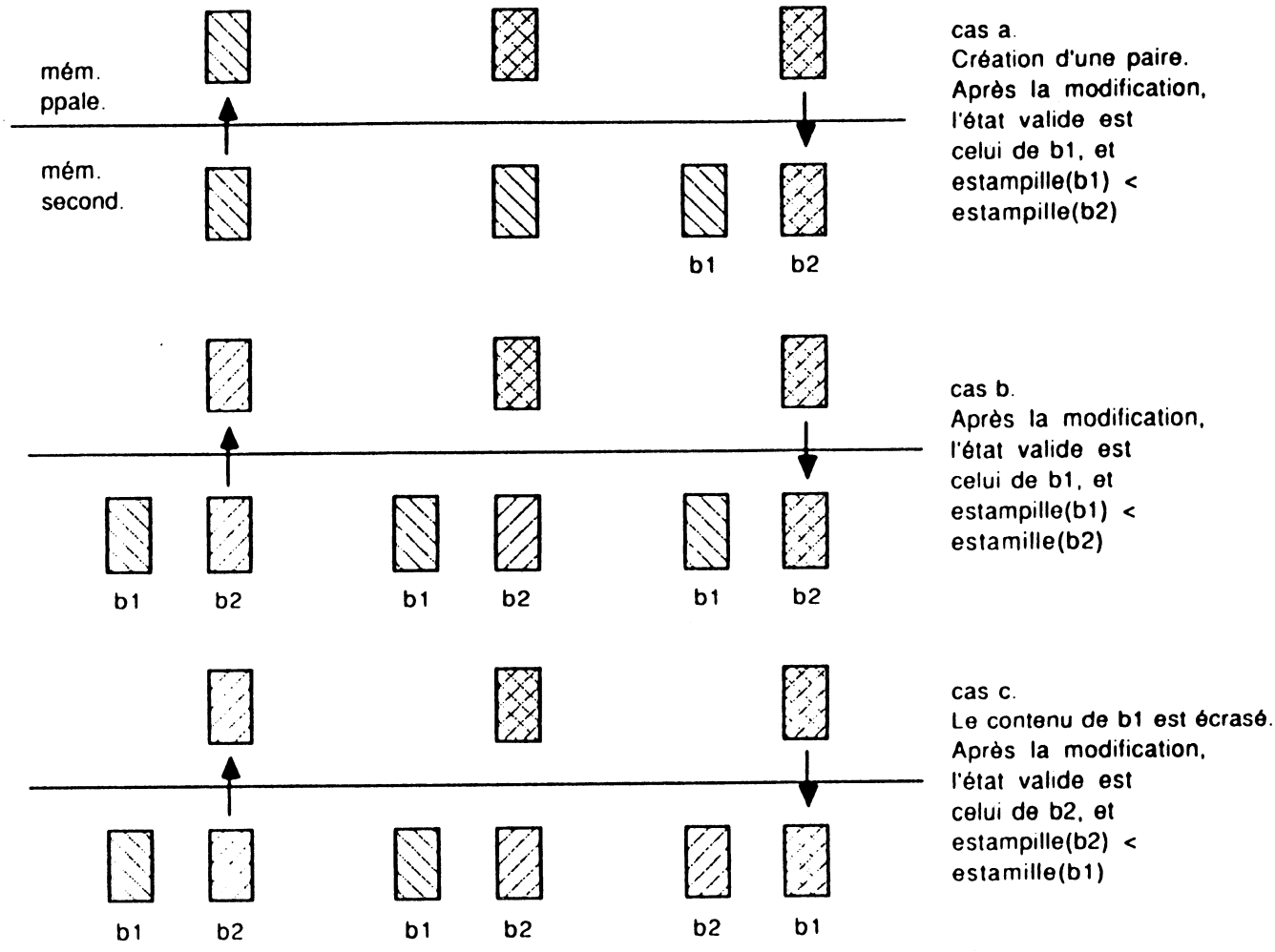


Figure 2.5 - Défaut de page et modification de pages virtuelles

virtuelle est modifiée, l'écriture sur disque ne se fait, dans aucun cas, sur le bloc qui contient l'ancien état valide mais sur l'autre. Ceci peut faire que les deux blocs de la paire soient échangés (cas c. de la figure 2.5).

Les actions réalisées pendant un point de contrôle sont les suivantes :

- Sauvegarde des registres du processeur (racine temporaire).
- Ecriture sur disque des pages 'sales' (modifiées) de mémoire virtuelle.
- Mise à jour de la date du dernier point de contrôle avec la date courante.

Les actions réalisées pendant la récupération après une panne du système sont les suivantes :

- Le système est ramené à l'état du dernier point de contrôle. Les registres du processeur sauvegardés à ce moment sont alors restaurés.

- Si l'état restauré dépend de la configuration du système au moment du dernier point de contrôle, il doit être mis à jour afin de l'accorder à la configuration actuelle (après la panne).
- Le système reprend son opération normale.

Les pannes des disques sont traitées de manière différente aux pannes du système. Ceci en raison du fait que les pannes des disques peuvent altérer le dernier état valide de la mémoire. Pour pallier cet inconvénient, le dernier état valide doit être sauvegardé sur un support différent. Après la panne d'un disque, le nouveau disque est initialisé à partir du dernier état sauvegardé.

A la suite d'une panne sont perdues les informations des pages modifiées depuis le dernier point de contrôle. En conséquence, plus les points de contrôle sont fréquents, moins il y aura de pertes. La perte d'une (petite) partie des objets peut être acceptable pour quelques applications mais certainement pas pour d'autres. Il faudrait, pour ces objets, un mécanisme plus élaboré de contrôle de cohérence (cf. section 2.2.2.2).

2.2.2.1.2 Récupération d'espace

Sur la *TI Explorer Lisp Machine*, les blocs-disque réservés pour la mise en oeuvre de la mémoire virtuelle sont groupés en ensembles de 16 blocs contigus. Le ramasse-miettes collecte automatiquement les pages virtuelles correspondantes aux ensembles de blocs qui ne contiennent que des miettes. Il rend ensuite les pages à l'allocateur de pages virtuelles et libère les blocs.

Dans la mémoire virtuelle dotée du mécanisme de maintien de cohérence, la libération de pages peut se faire aussitôt les miettes détectées. Il n'en est pas de même pour les blocs-disque. En effet, il se peut que quelques-uns de ces blocs appartiennent au dernier état valide et que, par conséquent, ils ne puissent pas être touchés avant le prochain point de contrôle. Thatte modifie donc le ramasse-miettes de manière à retarder la libération des blocs-disque contenant de miettes jusqu'à la réalisation du prochain point de contrôle.

Etant donné qu'à la suite d'une panne du système le contenu des registres du processeur se perd, les objets temporaires deviennent des miettes. Ils seront détruits lors du prochain passage du ramasse-miettes. En conséquence, seulement les objets persistants résistent aux pannes.

2.2.2.2 La gestion d'objets

La mémoire persitante est mise en oeuvre au-dessus de la mémoire virtuelle décrite dans la section précédente. Un objet est implanté sur une ou plusieurs pages contiguës de mémoire virtuelle. Une page peut comporter un ou plusieurs objets.

Une interface procédurale permet de consulter et de modifier la racine persistante. La manipulation directe de cette racine est défendue afin de la protéger. En effet, si elle

est détruite ou endommagée, il n'est plus possible de distinguer les objets persistants des temporaires.

La mémoire persistante est partagée par tous les processus qui s'exécutent sur la machine.

La gestion d'objets réalise deux services principaux :

1. Le service de noms.
2. La gestion des *log*s utilisés pour la mise en oeuvre des transactions.

Le service de noms crée et manipule des noms symboliques définis par les utilisateurs. Il réalise la correspondance entre ces noms et les objets persistants d'une machine. Ces noms sont des objets persistants.

Etant donné qu'il est persistant, un nom peut servir comme racine persistante des objets d'une application. Il est donc possible de diviser l'espace persistant en espaces disjoints ou non, selon les besoins des applications. La suppression du nom d'un objet consiste donc en le détachement de celui-ci de l'un de ces espaces mais non pas à sa destruction.

Les transactions garantissent que les modifications se font sur les objets indépendamment du mécanisme de maintien de cohérence. Deux objets persistants, les *log*s de *redo* et *undo*, sont utilisés pour faire avancer effectivement l'état du système au-delà du dernier point de contrôle. Toute opération de modification est enregistrée sur les deux *log*s.

Une transaction n'est validée (*committed*) que lorsque son *log* de *redo* est sauvegardé de manière fiable sur disque. Ainsi, même en face d'une panne du système, les modifications enregistrées sur ce *log* peuvent être refaites sur les objets. Les deux *log*s sont examinés lors de la récupération du système après une panne. Le *log* d'*undo* permet de défaire les actions faites par la transaction avant le dernier point de contrôle. Le *log* de *redo* ayant été stocké de manière à résister à la panne, il permet de réaliser effectivement les actions faites par la transaction.

2.2.2.3 Les applications de la mémoire persistante

La richesse structurale de la mémoire persistante lui permet de supporter des applications qui combinent la puissance des technologies d'intelligence artificielle et de bases de données. On peut citer les systèmes de CAO et les systèmes multi-media, les bases de connaissances et les bases de données à objets. Le premier prototype de la mémoire d'objets [Thatte86] a permis de comprendre l'utilité d'une telle mémoire pour ce type d'applications.

2.3 Etude comparative et conclusions

La conservation de l'information constitue le rôle capital de la mémoire secondaire. Elle se trouve, en effet, à la base de la persistance de l'information et fournit les moyens pour que celle-ci resiste, dans la mesure du possible, aux pannes du système. On remarque cependant l'importance qu'acquiert aussi la mémoire secondaire dans la mise en oeuvre des systèmes à grand nombre d'objets : Elle est l'instrument qui permet d'élargir, en la 'virtualisant', la mémoire principale. D'un point de vue conceptuel, au moins, la différence entre 'la mémoire secondaire pour implanter la mémoire virtuelle' et 'la mémoire secondaire pour implanter la mémoire d'objets' tend à disparaître. Quoi qu'on ait toujours la volonté de la voir comme un serveur, la mémoire secondaire est de plus en plus intégrée à ces systèmes.

De même que les mémoires de Thatte et de Decouchant, la mémoire d'objets de Guide est une réalisation du modèle de la mémoire persistante (cf. section 3.3). Il existe cependant une différence fondamentale entre ces trois réalisations : Tandis que Thatte se sert de la page comme 'unité d'élargissement' de la mémoire principale, Decouchant et Guide utilisent pour ceci l'objet. En effet, Thatte fait face aux défauts de page et les autres aux défauts d'objet. En plus, Thatte n'écrit sur disque que les pages 'sales' alors que Decouchant et Guide écrivent toutes les pages d'un objet modifié, même les 'propres'[7]. L'unité d'échange entre les deux niveaux étant l'objet, les mémoires de ces deux systèmes sont des mémoires virtuelles d'objets.

Puisque les pages des objets sont toutes écrites sur disque, le maintien de l'interface au niveau de l'objet rend très coûteuse la mise en oeuvre d'un mécanisme de points de contrôle. Ainsi, à la différence de celle de Thatte, les réalisations de Decouchant et Guide ne font pas de points de contrôle. Or, comme Thatte le remarque (cf. [Thatte86], pp. 153), l'écriture d'un objet sur disque ne suffit pas pour garantir la cohérence des rapports que celui-ci maintient avec les autres objets : les points de contrôle sont nécessaires. En conséquence, les mémoires d'objets de Decouchant et Guide peuvent présenter un certain risque d'incohérence.

Thatte fait de sorte que tous ses objets soient aussi cohérents que possible. Il introduit ensuite les transactions, afin que les utilisateurs puissent rendre plus robustes les objets. En revanche, les concepteurs du système Guide ont considéré qu'un seul niveau de cohérence suffisait, compte tenu, d'une part, des besoins des applications auxquelles s'adresse le système (cf. chapitre 3 et section 5.2) et, de l'autre, du coût que suppose la mise en place de ce double mécanisme. En conséquence, Guide ne fournit que la cohérence résultante de la définition de transactions (cf. section 3.2). Il laisse aux utilisateurs la liberté (ou la responsabilité) de déterminer les objets dont les opérations

---

7. Bien que, en principe, les objets Smalltalk-80 sont de taille réduite, quelques-uns des objets que Guide gère, comme les documents Grif [Quint86] dépassent amplement la taille d'une page de mémoire.

doivent se réaliser de façon atomique. Pour les autres objets, le système n'offre aucune garantie.

Dans les mémoires d'objets de Decouchant et de Guide se trouve implantée, d'une certaine manière, la notion de point de contrôle. Ceci a lieu lorsque tout le contenu de la mémoire principale doit être copié en mémoire secondaire pour la réalisation du ramasse-miettes (pour Guide, cf. section 4.4.2). La fréquence de cette opération n'est certes pas celle des points de contrôle de Thatte et l'on trouve toujours le besoin d'écrire toutes les pages de tous les objets et non seulement les 'sales'. La différence entre ces deux sortes de points de contrôle est conceptuelle : Decouchant et Guide copient tout pour faire le ramasse-miettes et non pour la cohérence. Thatte écrit ses pages sur disque pour diminuer le risque d'incohérence. Qu'il lui ait fallu rapporter le ramasse-miettes en mémoire secondaire au moment du point de contrôle n'est qu'une conséquence de son mécanisme de maintien de la cohérence. On peut constater d'ailleurs que son ramasse-miettes en mémoire principale n'a rien à voir avec le maintien de la cohérence.

Les systèmes à mémoire virtuelle paginée présentent, comme Kaehler le remarque (cf. [Kaehler86], pp. 92), un inconvénient : Puisque leur taille est, en principe, réduite, plusieurs objets tiennent sur une page ; puisqu'une page est chargée en mémoire principale lorsque l'un des objets qu'elle contient est appelé, la mémoire peut se saturer de pages sous-utilisées. Par contre, les systèmes à mémoire virtuelle d'objets sont capables de maintenir en mémoire principale seulement leur *working set*. Une étude comparative de ces deux tendances a permis a Stamos d'arriver aux conclusions suivantes [Stamos84] :

- Si les objets que les applications utilisent ensemble peuvent être regroupés en mémoire secondaire, les systèmes à mémoire virtuelle paginée et les systèmes à mémoire virtuelle d'objets fonctionnent correctement.
- Si les objets ne sont pas régroupés et si la taille de la mémoire principale approche celle du *working set*, les systèmes à mémoire virtuelle paginée et les systèmes à mémoire virtuelle d'objets ont la tendance à faire du *thrash*.

En conséquence, on doit regrouper en mémoire secondaire. Ni la réalisation de Thatte ni celle de Decouchant ne regroupent les objets. Cette fonction n'a pas non plus été considérée dans la première version de Guide, quoi que des études commencent pour l'intégrer dans une prochaîne version (cf. section Conclusions.2). Cette absence de regroupement contraste avec la richesse que dans ce domaine offrent Oracle, ObServer et l'OM/WiSS d'O_2. On peut également remarquer que, dans le cas des deux derniers systèmes, les objets ne sont pas seulement regroupés sur la mémoire secondaire du serveur mais ils sont aussi envoyés en groupe aux sites des clients.

En ce qui concerne la désignation par identificateurs, on constate que LOOM et ObServer possèdent deux espaces d'identificateurs séparés, correspondants aux deux niveaux de mémoire. Dans le gestionnaire de Decouchant, chaque site du réseau comporte un espace unique de désignation mais il n'existe pas d'espace global. Dans la réalisation de Thatte et dans le système Guide (cf. section 4.2), l'espace des

identificateurs est unique. Le nombre d'espaces se reflète logiquement dans le nombre de compteurs nécessaires pour la mise en oeuvre du ramasse-miettes :

- Le système LOOM utilise deux compteurs : l'un pour les objets en mémoire principale et l'autre pour les objets en mémoire secondaire.
- Le gestionnaire d'objets de Decouchant se sert aussi de deux compteurs : l'un pour les références externes et l'autre pour les locales.
- Les objets de Thatte et de Guide ne comportent qu'un seul compteur.
- ObServer lui-même ne fait pas de ramasse-miettes. Cette tâche incombe les systèmes à objets qui l'utilisent.

Deuxième partie : La gestion d'objets persistants dans le système Guide

3. Introduction au système Guide

Guide est un système d'exploitation réparti sur des postes de travail individuels et des serveurs spécialisés. L'objectif initial du projet Guide est de réaliser un système expérimental, qui doit servir de banc d'essai pour des recherches sur les systèmes répartis.

Les objectifs du système sont donc ceux de tout système d'exploitation réparti : Communication, accès aux informations distantes, partage d'information, stockage fiable d'information, utilisation d'outils de développement et mise au point d'applications.

Dans le cas de Guide, ces objectifs prennent en compte les aspects innovateurs des systèmes d'aujourd'hui : Nouvelles modalités de partage résultant de l'introduction de postes de travail puissants, nouvelles interfaces offertes aux usagers et nouveaux concepts de structuration de logiciel (modules, types abstraits, généricité).

Cependant, le trait le plus important du système est celui de l'**intégration**. Celle-ci a lieu dans deux directions :

1. Intégration de composants : Chaque poste de travail est considéré comme un composant d'un système d'exploitation global réparti. Les fonctions liées à la répartition ne sont pas surajoutées à un système déjà existant mais prises en compte dès l'origine.
2. Intégration de concepts en 'virtualisant' les ressources : Les ressources d'exécution et de conservation sont rendues virtuelles afin de décharger les utilisateurs de la nécessité de connaître leurs détails. Cette 'virtualisation' est faite de manière à favoriser une vue intégrée des composants du système.

Dans le but d'atteindre une vue intégrée du système, les concepteurs de Guide ont déterminé deux orientations de conception :

1. Introduction, dans l'organisation du système, des concepts des langages de programmation qui facilitent l'abstraction et la réutilisation de composants.
2. Incorporation, dans un ensemble cohérent, des aspects relatifs à l'exécution des programmes et de ceux qui concernent la conservation permanente de l'information.

Ainsi nous sommes-nous intéressés aux modèles à objets, en essayant d'y intégrer la conservation, sujet traité séparément jusqu'à présent. L'étude et la reflexion sur ces modèles ainsi que l'accent particulier mis sur l'intégration de la conservation d'objets nous ont permis de délimiter l'entité de base du système. Cette entité est l'**objet**, concept emprunté aux modèles analysés. Pour nous, un objet n'est pas seulement l'unité d'abstraction mais aussi, et en même temps, l'unité d'exécution et de conservation du système.

Le triple rôle d'un objet ressort des trois modèles en termes desquels a été finalement conçu le système. Ces modèles sont :

1. Le modèle de données. Il permet aux usagers d'abstraire, en utilisant des objets, la nature des données de leurs applications. Ce modèle est mis en oeuvre par le moyen d'un langage particulier, nommé 'langage Guide'.
2. Le modèle d'exécution. Il définit les environnements d'exécution du système. Un environnement d'exécution est constitué d'un ensemble d'objets et d'un ensemble de processus qui opèrent sur eux.
3. Le modèle de la mémoire. Il détermine les caractéristiques de l'espace d'exécution et de conservation qui comporte tous les objets du système. La mémoire du système Guide constitue une réalisation du modèle de la mémoire persistante de Thatte.

La superposition des modèles d'exécution et de mémoire met à la disposition des concepteurs d'applications une machine virtuelle multiprocesseurs dans laquelle le parallélisme est apparent et la distribution est cachée. Cette machine réalise donc les fonctions de la gestion d'objets. L'interface d'accès à la machine virtuelle est constituée par les instructions du langage Guide.

Ce chapitre présente une description générale du système Guide, en se concentrant sur deux des trois modèles qui ont déterminé ses spécifications actuelles. La conception et la mise en oeuvre du troisième modèle, celui de la mémoire, constituent les objectifs de ce travail de thèse. Elles seront présentées en détail dans les chapitres suivants.

Les modèles de données et d'exécution sont beaucoup plus riches qu'on ne le dit dans ce document. Certaines de leurs caractéristiques ne seront pas mentionnées car elles ne sont pas importantes en ce qui concerne la gestion de la mémoire, notre objectif principal. Une description approfondie du modèle de données est présentée dans [Guide-R2], [Guide-R7], [Krakowiak88] et [Meysembourg88]. Les détails du modèle d'exécution se trouvent dans [Guide-R3], [Guide-R5], [Decouchant88a] et [Decouchant88b].

3.1 Le modele de données

Un objet regroupe un ensemble de données, ou variables d'**état**, et un ensemble de procédures d'accès, appelées **méthodes**. L'état d'un objet ne peut être manipulé que par l'appel à une de ses méthodes.

3.1.1 Types et classes

Un **type** permet de rassembler les objets qui présentent une interface d'accès commune. Il est donc spécifié par l'ensemble des méthodes de cette interface. Une méthode est définie par une signature, qui décrit son nom et le type de ses paramètres, et par une spécification, qui décrit son effet.

Une **classe** spécifie la structure de l'état et les programmes (le code) des méthodes d'un objet. Elle décrit donc une réalisation particulière du type de l'objet. En conséquence, un type donné peut être réalisé par plusieurs classes différentes.

Les classes sont les constructeurs des objets. En effet, à chaque classe est associée une opération de génération qui permet de créer des objets ayant tous le même type, la même structure d'état et les mêmes programmes pour leurs méthodes. Ces objets sont appelés les **instances** (ou exemplaires) de la classe.

Toute classe est elle-même un objet et est donc manipulée comme tel par l'appel de méthodes spécifiques. En particulier, la méthode `New`, propre à toute classe, permet la génération d'instances.

Le modèle de données fournit un ensemble de types qui correspondent aux types 'de base' des langages traditionnels. Il s'agit des types `Boolean`, `Integer`, `Real`, `Char` et `String`. Le modèle de données prédéfinit aussi les classes qui réalisent ces types. Chaque classe a le même nom que le type qu'elle réalise.

On appelle objets **élémentaires** les instances de ces classes. Le traitement syntaxique que le langage Guide réserve à ces objets est le même que celui des langages traditionnels.

Le modèle de données pourvoit en outre les types prédéfinis génériques `Array` et `List`, ainsi que les classes qui les réalisent (de même nom que les types). Les instances de ces classes offrent aux usagers la possibilité de manipuler des tableaux et des listes d'objets de type quelconque.

Les types et classes prédéfinis sont les blocs de base qui permettent aux usagers de définir leurs propres types et classes. On appelle objets **construits** les instances de ces classes. En fait, l'état des objets construits est mis en oeuvre par groupement d'objets élémentaires et/ou d'instances des classes `Array` et `List`.

Les types et classes sont définis par le moyen des instructions et déclarations du langage Guide. L'exemple de la figure 3.1, programmé dans ce langage, permet d'illustrer les concepts précédents et d'en introduire d'autres que l'on expliquera par la suite.

```
TYPE Employee IS
METHOD Init (IN String ; Integer) : Integer ;
METHOD RaiseSalary (IN Integer) : Integer ;
END Employee.

CLASS Employee IMPLEMENTS Employee ;
name : String ; salary : Integer ;
```

```
METHOD Init (IN thatName : String ; thatSalary : Integer) : Integer ;
BEGIN
IF thatSalary <= 0
   THEN RETURN -1 ;
   ELSE salary := thatSalary ;
        name := thatName ;
        RETURN 0 ;
END ;
END Init ;

METHOD RaiseSalary (IN increase : Integer) : Integer ;
BEGIN
IF increase < 0
   THEN RETURN -1 ;
   ELSE salary := salary + increase ;
        RETURN 0 ;
END ;
END RaiseSalary ;

END Employee.

TYPE Management IS
METHOD Init ;
METHOD CreateEmployee (IN String ; Integer) : Integer ;
METHOD GetEmployee (IN Integer) : REF Employee ;
METHOD GoodEmployee (IN Integer) : Integer ;
END Management.

CLASS Management IMPLEMENTS Management ;
employees : LIST OF Employee ; current : Integer ;

METHOD Init ;
BEGIN
current := 0 ;
END Init ;

METHOD CreateEmployee (IN thatName : String ; thatSalary : Integer) : Integer ;
result : Integer ; employee : REF Employee ;
BEGIN
employee := Employee.New ;
result := employee.Init(thatName, thatSalary) ;
IF result < 0
   THEN RETURN result ;
   ELSE employees.Append(employee) ;
        // ajouter un élément en fin de liste
        current := employees.Size ;
        // obtention de la taille de la liste
        RETURN 0 ;
END ;
END CreateEmployee ;
```

```
METHOD GetEmployee (IN position : Integer) : REF Employee ;
BEGIN
IF position > employees.Size
   THEN RETURN NIL ;
   ELSE current := position ;
        RETURN employees.Go(current) ;
        // accès direct à un élément de la liste
END ;
END GetEmployee ;

METHOD GoodEmployee (IN increase : Integer) : Integer ;
BEGIN
RETURN (employees.Go(current)).RaiseSalary(increase) ;
END GoodEmployee ;

END Management.
```

Figure 3.1 - Exemple de types et classes

Une remarque concernant la création d'objets s'impose pour la bonne compréhension de la figure 3.1 : La méthode `New` ne s'exécute sur aucune classe prédéfinie. La création des instances de ces classes est prise en compte directement par le compilateur Guide.

Les instances des classes `Employee` et `Management` sont des objets construits. L'état d'une instance d'`Employee` comporte les objets élémentaires `name` et `salary`. L'état d'un objet de classe `Management` est constitué de `current`, un entier, et d'`employees`, une référence à un objet de classe `List` (une liste d'employés). Les références aux objets seront décrites ultérieurement.

La figure montre aussi comment sont définis les appels de méthode, seules manières d'accéder à l'état d'un objet : Il s'agit des expressions

```
reference_to_object.method_of_object(parameters)
```

L'appel particulier `class.New` permet de créer un nouvel objet, instance de la classe `class`. Enfin, par le biais de l'appel à la méthode `Init`, il est possible de donner une valeur initiale à l'état d'un objet.

La compilation d'une classe donne lieu à un objet binaire exécutable. Si le système Guide s'exécute sur un réseau hétérogène, à une classe peuvent correspondre plusieurs objets binaires exécutables. Les problèmes liés à l'hétérogénéité se trouvent pour l'instant en phase d'analyse, étant donné le peu de différences entre les systèmes-supports utilisés pour le développement actuel de Guide (cf. chapitre 6 et Conclusions). Pour ces raisons, on ne traite pas l'hétérogénéité dans ce document. On fait donc correspondre un seul objet binaire exécutable à une classe. La gestion d'un tel objet présente, comme l'on verra plus tard (cf. section 4.1.1.1), certaines caractéristiques particulières.

3.1.2 Persistance des objets

Le modèle de données permet à l'utilisateur de distinguer trois sortes d'objets selon leur durée de vie :

1. Les objets **de travail**. Ils sont définis à l'intérieur d'une méthode d'un objet englobant. Ils ne sont donc accessibles que par le biais des expressions du code de cette méthode. En conséquence, leur durée de vie est égale à celle de l'exécution de la méthode[8]. L'objet élémentaire `result` de la méthode `CreateEmployee` dans la figure 3.1 est un exemple d'objet de travail.
2. Les objets **internes**. Ils font partie de l'état d'un objet englobant et ils ne sont pas connus en dehors de cet objet. Leurs méthodes ne peuvent donc être appelées que par les méthodes de l'objet qui les contient. Par conséquent, leur durée de vie est égale à celle de cet objet. Un objet interne n'est visible qu'au niveau du langage. Dans la figure 3.1, les objets élémentaires `name` et `salary` sont des objets internes aux instances de la classe `Employee`.
3. Les objets **persistants** (ou 'permanents'). Ce sont des entités autonomes : Leur durée de vie est indépendante de tout autre objet ou de toute exécution de méthode. Ce sont les seuls objets pris en compte par les fonctions de gestion de la mémoire. Un objet persistant n'est détruit que par le mécanisme de ramasse-miettes (cf. section 4.4). Dans l'exemple de la figure 3.1, tous les employés, ainsi que la liste qui les regroupe, correspondent à des objets persistants.

Les objets élémentaires peuvent être de travail ou internes. Les objets construits peuvent être de travail, internes ou persistants. Tout objet binaire exécutable est persistant.

Le terme 'objet' sera utilisé désormais pour désigner indistinctement tout objet binaire exécutable et toute instance persistante d'une classe. En cas de nécessité seulement, la différence sera clairement établie. De même, l'objet binaire exécutable correspondant à une classe sera appelé simplement 'classe'.

3.1.3 Désignation des objets

Tout objet du système est identifié de façon unique à l'aide d'un identificateur universel nommé **oid** (*object identifier*). Les oid's sont attribués lors de la création des objets et ne sont jamais modifiés.

Une **référence** permet de désigner et de manipuler un objet indépendamment de tout mécanisme de gestion de mémoire à l'exécution. Une référence comporte l'oid d'un objet et quelques informations qui permettent de le localiser à l'intérieur de la

8. C'est pourquoi ils sont aussi appelés objets 'temporaires'.

mémoire. Le contenu d'un oid et celui d'une référence seront détaillés ultérieurement (cf. section 4.2). Plusieurs références peuvent désigner le même objet.

Le mot réservé `REF` du langage Guide permet de déclarer des références à l'intérieur du code d'une classe. Les valeurs des références résultent des affectations et des appels et retours de méthodes. En particulier, la méthode `New` de toute classe retourne une référence sur l'objet qu'elle crée. Cependant, le contenu des références n'est pas directement accessible en utilisant les instructions du langage.

La figure 3.1 illustre également la manipulation de références. Dans la méthode `CreateEmployee`, par exemple, la référence `employee` est utilisée pour désigner une instance de la classe `Employee`. L'objet est créé et inséré dans la liste d'employés et, à la fin de la méthode, sa référence est détruite. La référence `employees`, qui fait partie de l'état des instances de `Management`, permet l'appel de toutes les méthodes (pré)définies dans la classe `List`. La plupart de ces méthodes retournent des références.

Les références sont utilisées de plusieurs manières :

- Pour appeler les méthodes des objets.
- Pour passer en paramètre de méthode un objet construit : On passe une référence vers cet objet.
- Pour construire des objets composés (cf. section 3.1.4).
- Pour localiser des objets (cf. section 4.3).
- Pour désigner les objets à l'intérieur des fonctions de gestion qui mettent en oeuvre les modèles d'exécution et de mémoire (cf. chapitres 4 et 6).

D'autres manières de désignation d'objets sont mises en oeuvre par deux des services du système Guide. Il s'agit des services de désignation symbolique et de de gestion de versions (cf. chapitre 5).

3.1.4 Composition d'objets

Un objet persistant est dit **simple** s'il ne contient pas (si son état ne comporte pas) de références sur d'autres objets persistants. Un objet est **composé**, par contre, s'il contient au moins une référence sur un autre objet persistant, qui peut lui-même être simple ou composé. Les instances de la classe `Employee` de la figure 3.1 sont des objets simples. Celles de la classe `Management` sont des objets composés.

Le mécanisme de composition d'objets permet, dans la plupart des cas, d'éviter les très grands objets. Considérons, par exemple, un document qui comporte plusieurs chapitres. Deux réalisations sont possibles en langage Guide :

1. L'état de l'objet document contient tous les chapitres.
2. L'objet document ne comporte que des références à des objets réalisant les chapitres.

Dans le deuxième cas, la taille de l'objet document est considérablement plus petite que celle qu'il aurait dans le premier cas.

La composition d'objets est aussi à la base du partage d'objets, un même objet pouvant être un composant de plusieurs objets composés. C'est le cas, par exemple, des étudiants qui suivent plus d'un cours. Si les objets cours sont composés des objets correspondant aux étudiants, l'objet correspondant à un étudiant qui assiste à plus d'un cours est composant de plusieurs objets cours.

On appelle **composition d'objets issue d'un objet** l'ensemble des objets contenant la fermeture transitive de la relation 'possède une référence vers', fermeture construite en partant de cet objet. Le système Guide fournit un objet particulier dont la composition englobe la totalité des objets persistants. Il s'agit de la racine persistante du système.

A l'intérieur de l'état d'un objet, les références vers d'autres objets sont regroupées dans une table appelée **table de références**. Elle est créée par le compilateur et est mise à jour au fur et à mesure des affectations des références. Tout objet comporte aussi un compteur interne appelé le **compteur de références**. La valeur de ce compteur est le nombre d'objets qui référencent l'objet englobant.

La table de références et le compteur de références d'un objet sont accessibles par la gestion d'objets indépendamment de l'exécution des méthodes de l'objet. Ceci rend possible la réalisation de mécanismes de gestion prenant en compte la composition d'objets, comme la récupération d'espace (cf. section 4.4).

A l'exception de ces mécanismes, la gestion d'objets n'est concernée que par des objets individuels. La prise en compte de la composition doit donc être effectuée par les applications. Dans la suite, quand on parle d'un objet, on désigne uniquement cet objet et non la composition d'objets issue de cet objet.

3.2 Le modèle d'exécution

L'objectif du modèle d'exécution de Guide est fournir aux utilisateurs les structures nécessaires pour l'exécution de leurs applications. Cet objectif peut être accompli de trois manières :

1. Faire des objets des entités 'actives', c'est-à-dire, les rendre multi-processus. Les méthodes d'un objet deviennent des requêtes que celui-ci fait exécuter par ses processus internes. L'objet se comporte donc comme un serveur.
2. Faire des objets des entités 'passives', c'est-à-dire, leur ôter toute possibilité d'exécution, et définir en même temps des environnements d'exécution multi-processus qui se chargent de l'exécution des méthodes des objets.
3. Combiner les deux possibilités ci-dessus.

Deux raisons nous ont amenés à choisir la deuxième voie : On a préféré que les applications se structurent en termes d'appels de services et non pas d'appels à un

serveur. Il n'y a pas de raison objective d'intégrer chaque service dans un serveur. Ce choix nous a permis en outre de garder un structure légère tant pour les objets que pour les processus.

Un environnement d'exécution est constitué d'un ensemble d'objets et d'un ensemble de processus exécutant les méthodes de ces objets. Par analogie avec les systèmes à capacités, les environnements d'exécution sont appelées **domaines**. Les processus, vraies entités actives, sont nommés **activités**.

L'ensemble des objets qui composent un domaine à un instant donné est appelé son **contexte**. La composition du contexte évolue dans le temps par le moyen des opérations de **liaison** et de **déliaison** dynamiques. Le contexte d'un domaine est matérialisé par une structure de données appelée **descriptif**.

Un objet peut être partagé entre plusieurs domaines. Dans ce cas, il possède une entrée dans le descriptif de chacun. Le partage d'objets entre domaines est montré dans la figure 3.2. En revanche, une activité ne peut appartenir qu'à un seul domaine.

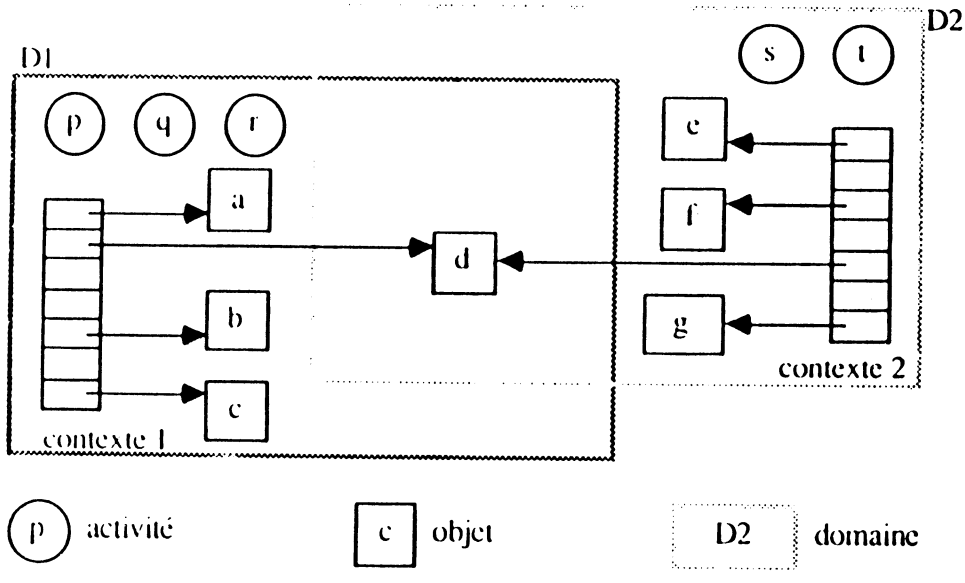


Figure 3.2 - Domaines, activités et partage d'objets

Les activités d'un domaine s'exécutent en parallèle. Elles communiquent à travers les objets du domaine, protégés, si besoin est, par des mécanismes de synchronisation. Le partage d'objets est donc à la base de la communication et la synchronisation entre activités. Le contrôle d'accès aux objets partagés s'exprime à l'aide de **conditions de synchronisation et d'exécution** associées à leurs méthodes. Ces mécanismes permettent aussi le partage d'un objet entre activités de différents domaines.

La cohérence de l'exécution d'un ensemble de méthodes est garantie par le concept de **transaction**. Une transaction est une séquence d'exécution atomique. Elle est donc associée à une séquence de méthodes d'objet exécutées dans un domaine [Ledot88]. La

possibilité de lancer des activités parallèles permet d'introduire les transactions imbriquées.

3.2.1 Aspects dynamiques du modèle

Un domaine est créé par une activité d'un autre domaine. L'opération de création spécifie un objet initial et une méthode de cet objet. Une activité initiale est créée pour exécuter la méthode sur l'objet initial. Elle engendre, s'il y a lieu, les autres activités du domaine. Une fois créé, un domaine est autonome, mais l'activité qui le crée peut exercer un certain contrôle sur lui. Elle peut, en particulier, le détruire, l'arrêter et le faire reprendre son déroulement.

La composition d'un domaine évolue en fonction des appels de méthodes exécutés par ses activités. Il y a liaison dynamique d'un objet dans le domaine lorsqu'une activité appelle une méthode de l'objet (on dit aussi que l'activité 'veut exécuter une méthode de l'objet') et que celui-ci ne fait pas partie du contexte du domaine. L'objet est donc recherché dans la mémoire d'objets (cf. section 4.3) et une entrée lui est allouée dans le descriptif du domaine.

La déliaison d'un objet d'un domaine consiste à le retirer du contexte du domaine. Lorsqu'une activité du domaine ne souhaite plus utiliser l'objet, elle peut le délier. Elle ne peut cependant le faire que si aucune autre activité du domaine n'execute une méthode de l'objet. A la terminaison du domaine, les objets encore liés sont déliés.

Une activité d'un domaine peut créer une nouvelle activité. On appelle ces activités, respectivement, la **mère** et la **fille**. La mère désigne un objet initial et la méthode de cet objet que la fille doit exécuter. La fille appartient au même domaine que sa mère.

Une activité crée des filles pour les faire exécuter en parallèle des méthodes des objets de son domaine. Le langage Guide permet de déterminer ces méthodes par le biais du bloc suivant :

```
CO_BEGIN
tag_1 : reference_P_to_object.method(parameters)
        .......
tag_N : reference_Q_to_object.method(parameters)
CO_END defined-on-tags_condition.
```

Les N filles exécutent en parallèle les méthodes indiquées mais la mère est suspendue. Cette restriction est imposée par le contrôle des accès concurrents aux objets lorsque le bloc est mis en oeuvre dans le contexte d'une transaction. La reprise de la mère a lieu lorsque la condition du `CO_END` est vraie. Cette condition porte sur la terminaison ou l'abandon des filles. Un nouveau bloc `CO_BEGIN-CO_END` peut évidemment être mis en oeuvre par l'une des filles.

Le bloc `CO_BEGIN-CO_END` est le seul moyen que le langage Guide offre aux usagers pour la manipulation d'activités. Ceci découle du souci de la programmation des applications en termes d'objets et non pas de processus (cf. ci-dessus).

Une activité se termine lorsqu'elle exécute l'instruction de retour de la méthode de l'objet initial. Un domaine se termine lorsque se termine son activité initiale.

Un souci d'uniformité nous a amenés à considérer les domaines et les activités comme des objets Guide particuliers. Leurs types et classes sont prédéfinis et soumis à des restrictions d'accès [Guide-R1] [Guide-R3] [Guide-R5].

3.2.2 Vue des objets selon le modèle d'exécution

Le modèle d'exécution permet aux utilisateurs de distinguer trois sortes d'objets selon la manière d'exécution de leurs méthodes : Les objets **synchronisés**, les objets **atomiques** et les objets **ordinaires**.

Un objet est synchronisé si, dans le code de sa classe, des conditions de synchronisation entre ses méthodes ont été définies. Toutes les instances d'une telle classe sont des objets synchronisés.

Un objet est atomique s'il ne peut être modifié qu'à l'intérieur d'une transaction. Il peut, par contre, être utilisé en consultation soit à l'intérieur, soit en dehors d'une transaction. Dans ce dernier cas, il peut y avoir incohérence des données consultées. Un objet est déclaré atomique au moment de sa création. L'atomicité d'un objet est donc indépendante de la classe de l'objet.

Un objet est ordinaire s'il n'est ni synchronisé ni atomique. Les appels de méthode sur un objet ordinaire ne sont soumis à aucun contrôle, hormis ceux des mécanismes de protection [Guide-R4].

Les notions d'objet atomique et d'objet synchronisé sont indépendantes, même si dans la pratique la plupart des objets atomiques sont synchronisés. Le fait qu'un objet composé soit atomique n'implique pas que les objets composants le soient aussi.

3.3 Le modèle de la mémoire

L'objectif du modèle de la mémoire est de caractériser le support d'exécution et de conservation des objets du système.

Afin d'exécuter une méthode d'un objet, une activité doit pouvoir désigner cet objet dans son espace d'adressage. Ceci est possible dès que l'activité couple (attache) l'espace en mémoire de l'objet dans son propre espace d'adressage. Ensuite, une fois qu'elle n'utilise plus l'objet, l'activité le détache de son espace d'adressage.

Pour aboutir à une mémoire d'objets qui donne un tel support aux activités, nous avons exploré deux modèles *conceptuellement* très différents mais dont les réalisations, surtout en milieu distribué, peuvent être très proches. Ces modèles sont :

1. Tous les objets sont conservés dans un espace global unique et chaque espace d'exécution (le contexte d'un domaine, dans le cas de Guide) est une fenêtre sur cet espace global. Ce modèle n'est qu'une généralisation d'une mémoire globale

segmentée à la Multics, chaque objet étant contenu dans un segment [Organick72]. Une réalisation répartie en est fournie par Apollo/Domain [Nelson83].

2. Les espaces d'exécution et de conservation sont indépendants, comportant chacun ses propres mécanismes de désignation. Les transferts d'objets entre les deux espaces sont explicites. En fait, l'espace de conservation se comporte comme un serveur d'objets vis-à-vis de chaque espace d'exécution.

Le modèle de l'espace global réalise par rapport à celui du serveur d'objets une économie *conceptuelle* : Il n'y a pas de transferts explicites d'objets. Il a été adopté comme modèle de la mémoire. Par conséquent, la mémoire de Guide est un espace unique d'exécution et de conservation qui comporte tous les objets du système. Les domaines sont des fenêtres sur cet espace.

En plus du support donné aux activités, le modèle de la mémoire doit prendre en compte la persistance des objets. La mémoire du système Guide garantit la persistance de ses objets de la manière suivante : Une fois créés, les objets sont conservés en mémoire jusqu'à leur destruction par un mécanisme de ramasse-miettes.

En tant qu'espace global d'objets doté de ramasse-miettes, la mémoire du système Guide est une réalisation du modèle de mémoire persistante de Thatte. Les cinq règles pour le maintien de l'intégrité de la fermeture transitive de la racine persistante (cf. Introduction) sont respectées : Les trois premières sont assurées du fait que le langage Guide est un langage de haut niveau traduit en C. L'existence même du ramasse-miettes satisfait la quatrième condition. La cinquième est garantie parce que la gestion de la mémoire se fait sans avoir à répondre aux requêtes des utilisateurs. Pour eux, en effet, l'espace de mémoire est une ressource que le système Guide a 'virtualisée'.

En principe, toute modification à un objet est incorporée dans son état de manière cohérente. Une cohérence forte est assurée seulement pour les objets atomiques manipulés au cours d'une transaction [Ledot88].

Les fonctions principales de la gestion de la mémoire sont la création, la destruction et la localisation des objets dans l'espace global. Ces fonctions se trouvent à la base de la gestion d'objets et constituent la clef de voûte de ce document. On n'approfondit plus ici sur le modèle de la mémoire et sa réalisation : Les chapitres 4 et 6 les décrivent en détail. D'autres présentations se trouvent dans [Guide-R4] et [Guide-R6].

3.4 La machine virtuelle de Guide

La superposition des modèles d'exécution et de la mémoire permet de définir la machine abstraite du langage Guide. Elle constitue donc le support de réalisation des applications définies en termes du modèle de données. Il s'agit d'une machine virtuelle multiprocesseurs qui masque la nature répartie du système mais qui fait ressortir le parallélisme d'exécution des activités.

L'architecture de la machine virtuelle, illustrée dans la figure 3.3, comporte deux niveaux d'abstraction. Le premier, appelé la **machine à objets**, est construit au-dessus des ressources physiques. Cette machine est constituée d'un ensemble de processeurs virtuels qui partagent une mémoire contenant tous les objets du système. Le niveau supérieur de la machine virtuelle met en oeuvre les domaines et les activités.

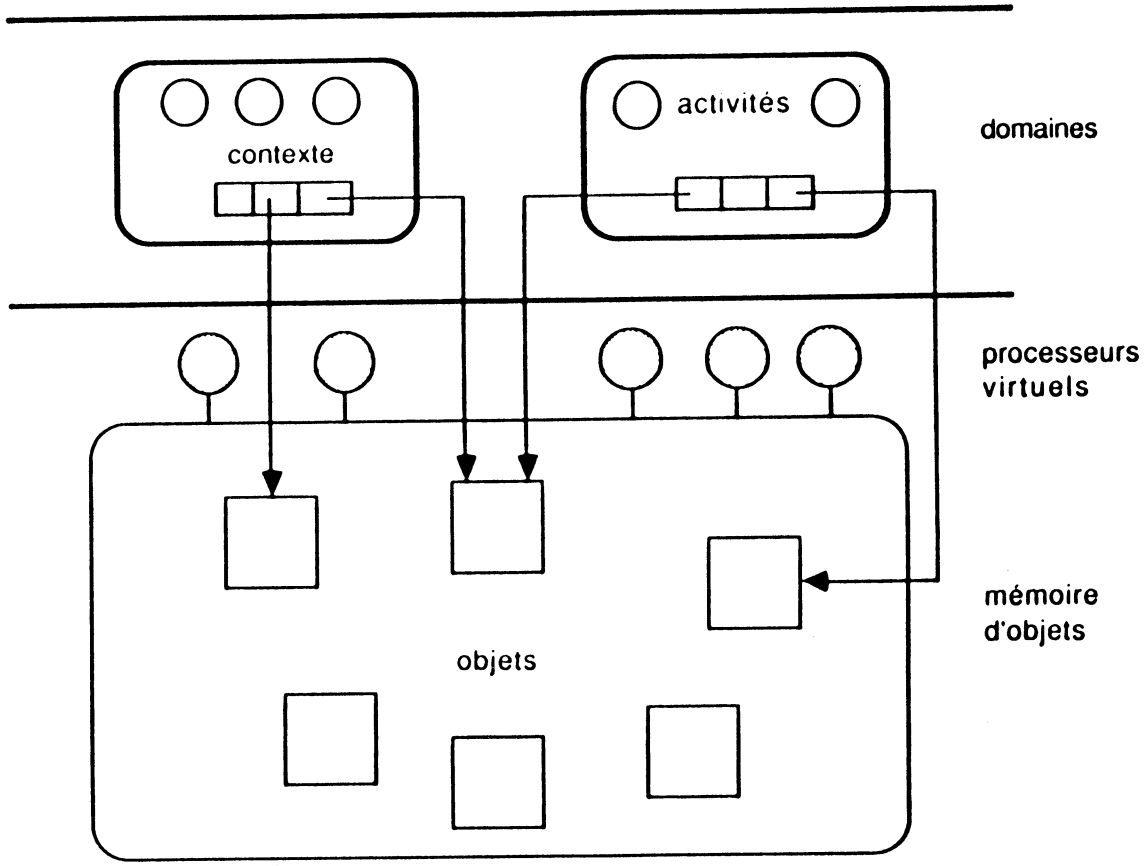


Figure 3.3 - La machine virtuelle de Guide

Du point de vue des applications, les instructions du langage Guide suffisent comme interface de la machine virtuelle. Cependant, certains services du système requièrent une souplesse plus grande dans la définition des entités visibles et des procédures de manipulation de ces entités. C'est le cas d'un service de mise au point de programmes (*debugger*) écrits en langage Guide ou d'un service d'intégration efficace de pilotes d'unités d'entrées/sorties (*drivers*) physiques.

Ces considérations ont amené les concepteurs du système à définir une interface plus riche et plus souple que celle qu'offrent les instructions du langage Guide. Cette interface constitue le jeu de primitives de la machine virtuelle. Le compilateur traduit les expressions du langage dans des appels à ces primitives.

3.4.1 La machine à objets

Un processeur virtuel est une machine qui interprète des appels de méthodes sur des objets. L'ensemble d'objets qu'un processeur virtuel peut adresser représente une fenêtre sur la mémoire d'objets. L'état d'un processeur virtuel est défini par les structures suivantes :

- La pile des contextes d'exécution correspondants aux appels de méthodes en cours. Un contexte d'exécution comporte une référence à l'objet dont l'une des méthodes est appelée et le nom et les paramètres de cette méthode.
- Un registre qui maintient l'adresse de l'état de l'objet dont l'une des méthodes est exécutée couramment. On appelle 'courant' cet objet.
- Un registre qui maintient l'adresse du code de la méthode exécutée couramment.

La mémoire d'objets est une mémoire segmentée de capacité potentiellement illimitée et dans laquelle l'unité d'allocation et d'adressage est l'objet. Elle constitue le support d'exécution et de conservation des objets. Elle contient tous les objets du système.

La machine à objets réalise la gestion des processeurs virtuels et la gestion de la mémoire d'objets. La gestion des processeurs virtuels consiste, d'une part, en l'allocation, la libération et la synchronisation des processeurs et, de l'autre, en l'interprétation des appels de méthode sur les objets. La gestion de la mémoire consiste en la création, la destruction et en la localisation d'objets dans l'espace de la mémoire.

Ces deux groupes de fonctions sont mis en oeuvre par l'ensemble de primitives de l'interface de la machine à objets. Cette interface est décrite dans [Guide-R3]. Les deux primitives dans lesquelles la gestion de la mémoire d'objets joue un rôle sont les suivantes :

1. `CallNewContext`
   Cette primitive permet l'interprétation, par un processeur virtuel, d'un appel de méthode sur un objet. La gestion de la mémoire localise l'objet et le rend adressable par le processeur virtuel.
2. `CreateObject`
   Cette primitive permet la création, par un processeur virtuel, d'un objet en tant qu'instance d'une classe. La gestion de la mémoire crée l'objet en mémoire (lui réserve de l'espace et l'initialise) et le rend adressable par le processeur virtuel.

3.4.2 Le gestionnaire des domaines et des activités

La gestion de domaines et d'activités constitue la couche haute de la machine virtuelle. Elle se charge de mettre en oeuvre les domaines et les activités au-dessus de la machine à objets.

A chaque activité d'un domaine est associé un processeur virtuel de la machine à objets. En conséquence, la création d'une nouvelle activité à la suite d'un appel de méthode sur un objet se traduit par l'allocation d'un processeur virtuel qui se charge d'interpréter l'appel. L'ensemble d'objets qu'une activité attache dans son espace d'adressage au cours de son exécution corresponde donc à la fenêtre sur la mémoire d'objets du processeur virtuel associé (cf. ci-dessus).

L'union des fenêtres sur la mémoire d'objets que comportent les processeurs virtuels associés aux activités d'un domaine donne lieu à la mémoire commune du domaine. Cette mémoire commune n'est autre chose que le contexte du domaine.

La gestion de domaines et d'activités comporte les groupes de fonctions suivants :

- La création, la manipulation et la destruction de domaines.
- La création, la manipulation, la synchronisation et la destruction d'activités.
- L'exécution, par les activités, des méthodes des objets.

Ces trois groupes de fonctions sont mis en oeuvre par l'ensemble de primitives de l'interface du gestionnaire des domaines et des activités. Puisque ce gestionnaire compose la couche haute de la machine virtuelle, son interface est aussi celle de cette machine. Cette interface est décrite dans [Guide-R3].

Les primitives qui mettent en oeuvre l'exécution d'une méthode d'objet à la suite d'un appel sont précisément celles qui appellent les deux primitives de la machine à objets dans lesquelles la gestion de la mémoire d'objets joue un rôle. Il s'agit des deux primitives suivantes :

1. `guideCall`
   L'execution de cette primitive permet à une activité de réaliser l'appel à une des méthodes d'un objet. Deux appels successifs à la primitive `CallNewContext` rendent adressables l'objet et sa classe par le processeur virtuel associé à l'activité. Ces deux objets sont liés dans le domaine de l'activité avant que le processeur n'interprète l'appel à la méthode. Après l'exécution de celle-ci, les deux objets sont déliés du domaine.
2. `guideCreate`
   L'execution de cette primitive permet à une activité de réaliser l'appel à la méthode `New` d'une classe. Par le biais de cette primitive, l'activité crée donc une instance de la classe. L'appel à la primitive `CallNewContext` rend adressable la classe par le processeur virtuel associé à l'activité. La classe est liée ensuite dans le domaine de l'activité. Enfin, l'instance est créée par l'appel à la primitive `CreateObject` de la machine à objets.

3.5 Les services du système Guide

Un service du système est une fonction de gestion d'objets que Guide, en tant que système spécialisé ou d'après certains de ses objectifs, doit fournir mais qui n'a pas été

intégrée comme fonction de la machine virtuelle.

Les services du système n'ont pas été intégrés dans la machine virtuelle pour les raisons suivantes :

- Les fonctions des services ne sont pas essentielles pour le bon déroulement du système. En d'autres termes, leur absence n'empêche pas que Guide n'accomplisse ses objectifs, tout au moins les principaux.
- Les services ne sont pas utilisés par toutes les applications. Il ne sont pas exhaustivement utilisés.
- La réalisation de la machine virtuelle doit rester aussi légère et souple que possible.
- Les fonctions de certains services constituent d'excellents moyens de validation et d'exploitation de la réalisation de la machine virtuelle.

Les deux dernières raisons sont surtout valables pour les premières réalisations du système [Guide-R5] [Decouchant88b].

Les fonctions suivantes constituent des services du système :

- L'administration d'objets : Définition et mise en place de critères de création, exécution des méthodes et conservation des objets (cf. section 4.1.2.1).
- La manipulation, comme un tout, de la composition issue d'un objet.
- La désignation symbolique d'objets.
- La création et manipulation de versions d'objets.
- L'exploitation de l'information interne aux types et classes. Ceci serait utilisé pour accéder aux objets par le contenu (accès associatif).
- La mise au point de programmes écrits en langage Guide.
- L'interface d'utilisation du système (interface homme-machine).

D'autres services peuvent évidemment être spécifiés.

Il existe deux manières de réalisation des services : La programmation en langage Guide et l'utilisation directe des primitives de la machine virtuelle. Le choix dépend du degré de souplesse requis par les entités qui mettent en oeuvre le service. Deux exemples de service réalisés en utilisant les primitives de la machine virtuelle ont déjà été cités (cf. section 3.4). La gestion de noms et la gestion de versions sont des exemples de service mis en oeuvre en langage Guide.

Les services réalisés en langage Guide sont conçus et s'exécutent comme des applications Guide. En conséquence, deux solutions sont possibles pour les spécifier :

1. Les fonctions du service sont mises en oeuvre par les objets et activités d'un domaine indépendant des autres domaines. Ce domaine se comporte donc comme un serveur.

2. Les fonctions du service sont réalisées par les méthodes d'un ensemble d'objets spécialisés qui sont partagés par tous les domaines.

Ces deux solutions sont présentées et comparées ci-dessous. Une étude plus approfondie se trouve dans [Freyssinet87].

Dans la solution serveur, un mécanisme de communication entre les domaines clients et le domaine serveur est nécessaire. Ceci peut être effectué à l'aide d'un objet interface, partagé entre chaque domaine client et le domaine serveur. Le service s'effectue alors par le moyen des méthodes de cet objet interface, qui fonctionne en producteur-consommateur. Le service lui-même est réalisé par les objets du domaine serveur qui prennent en compte les requêtes déposées dans les objets interfaces.

Dans la deuxième solution, le service est réalisé par l'appel des méthodes des objets spécialisés. Les activités des domaines clients lient ces objets dans leurs domaines et en invoquent les méthodes pour la réalisation du service. Le service est donc relatif à l'objet sur lequel on exécute la méthode. Dans certains cas, la référence de cet objet pourrait être conservée dans le contexte d'exécution de l'activité qui requiert le service.

La solution serveur a l'avantage de permettre un contrôle plus rigoureux sur les données de réalisation du service. Elle nécessite cependant un mécanisme de communication spécifique via des objets interfaces, ce qui risque de pénaliser les performances.

La solution service a l'avantage de la simplicité mais peut mettre en oeuvre un grand nombre d'objets qui sont partagés entre de nombreux domaines. Elle ne nécessite pas de mécanismes particuliers pour la communication entre domaines.

La solution service a été retenue pour la première réalisation des services de désignation symbolique et de gestion de versions (cf. chapitre 5).

4. Le modèle de la mémoire

Le modèle de la mémoire est celui d'un espace global unique qui contient tous les objets. Les domaines sont des fenêtres sur cet espace. Les objets ayant des tailles différentes, l'espace de la mémoire se présente comme une généralisation d'une mémoire segmentée. Les objets sont conservés en mémoire tant qu'ils ne sont pas ramassés.

La description des fonctions de gestion de la mémoire permet de mettre en évidence les rapports existants entre celle-ci et les autres composants de la machine à objets. Ceci est précisément l'objectif des sections suivantes.

4.1 Description de la mémoire d'objets

Les objets sont en même temps les unités d'exécution et de conservation du système. De ce fait résulte la double fonction que doit remplir la mémoire : Supporter l'exécution des méthodes des objets et garantir la conservation des objets jusqu'à leur destruction par le ramasse-miettes.

Le support d'exécution consiste, d'une part, à permettre et même faciliter aux entités du modèle d'exécution l'accès aux objets pour l'exécution de leurs méthodes et, de l'autre, à prendre en compte les conséquences de ces exécutions. Il se peut, en effet, que l'état d'un objet soit modifié ou même que sa taille change à la suite de l'exécution d'une de ses méthodes.

Le fait qu'un objet soit persistant n'implique pas qu'une fois créé, il est simplement et indéfiniment conservé en mémoire. Celle-ci doit le tenir prêt à toute manipulation, en particulier à celles qu'entraîne l'éventuelle exécution d'une de ses méthodes. De même, la mémoire doit intégrer dans son état, de manière permanente, les modifications qui résultent de cette exécution.

Etant donné qu'une exécution de méthode ne peut avoir lieu que sur une mémoire principale et que la conservation ne peut être garantie que par une mémoire secondaire, la mémoire d'objets doit être découpée en deux niveaux :

1. Le support des objets en mémoire principale, ou **Mémoire Virtuelle d'Objets** (MVO), qui permet l'exécution des méthodes.
2. Le support des objets en mémoire secondaire, ou **Mémoire Permanente d'Objets** (MPO), qui réalise la conservation.

La figure 4.1 illustre les deux niveaux de la mémoire.

Avec ces deux niveaux, le terme 'objet' désigne désormais un concept abstrait, correspondant en réalité à deux entités différentes : La représentation de l'objet en MVO, qu'on appelle **image**, et la représentation de l'objet en MPO, appelée **objet stocké**. Sur une image s'exécutent effectivement les méthodes de l'objet. Etant rangé en

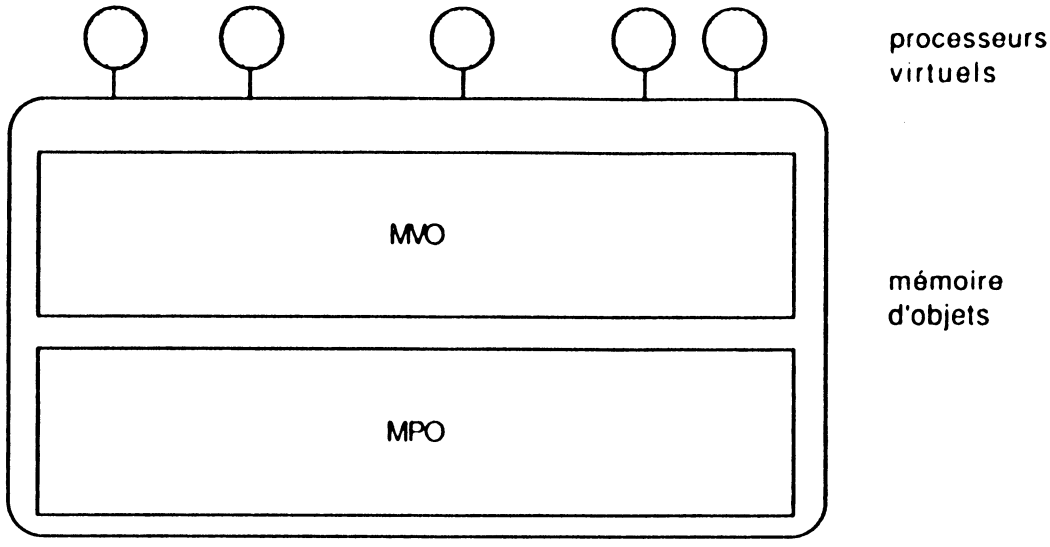


Figure 4.1 - La MVO et la MPO à l'intérieur de la machine à objets

mémoire secondaire, un objet stocké conserve l'état de l'objet correspondant.

Lorsqu'une activité appelle une des méthodes définies sur un objet, une image est créée en chargeant en MVO l'objet stocké correspondant. L'image reste en MVO pendant l'exécution seulement ou jusqu'à ce que, pour des raisons qu'on analysera plus tard (cf. section 4.4), on décide de la détruire. Auparavant, si l'image a été modifiée à la suite de l'exécution de la méthode, elle doit être rangée en MPO de manière à remplacer l'ancien contenu de l'objet stocké. Celui-ci n'est pas détruit, ce qui garantit que l'objet correspondant est conservé.

La figure 4.2 montre les représentations d'un objet et les échanges qui ont lieu entre la MVO et la MPO pour leur mise en place.

Le chargement d'un objet stocké et le rangement d'une image constituent la première approche aux interactions entre la MVO et la MPO. Cette interface est *interne* à la gestion de la mémoire. Elle n'est donc pas visible par les autres niveaux de la machine à objets.

4.1.1 Description des objets en MVO

A l'intérieur de la machine à objets, chaque processeur virtuel dispose d'une mémoire virtuelle qui définit les objets qu'il peut adresser pour l'exécution de méthodes. Pour qu'un objet puisse être adressable par un processeur virtuel, son image doit être présente dans la mémoire virtuelle de celui-ci. En conséquence, la composition de cette mémoire virtuelle évolue dynamiquement en fonction des primitives `CallNewContext` et `CreateObject` de la machine à objets.

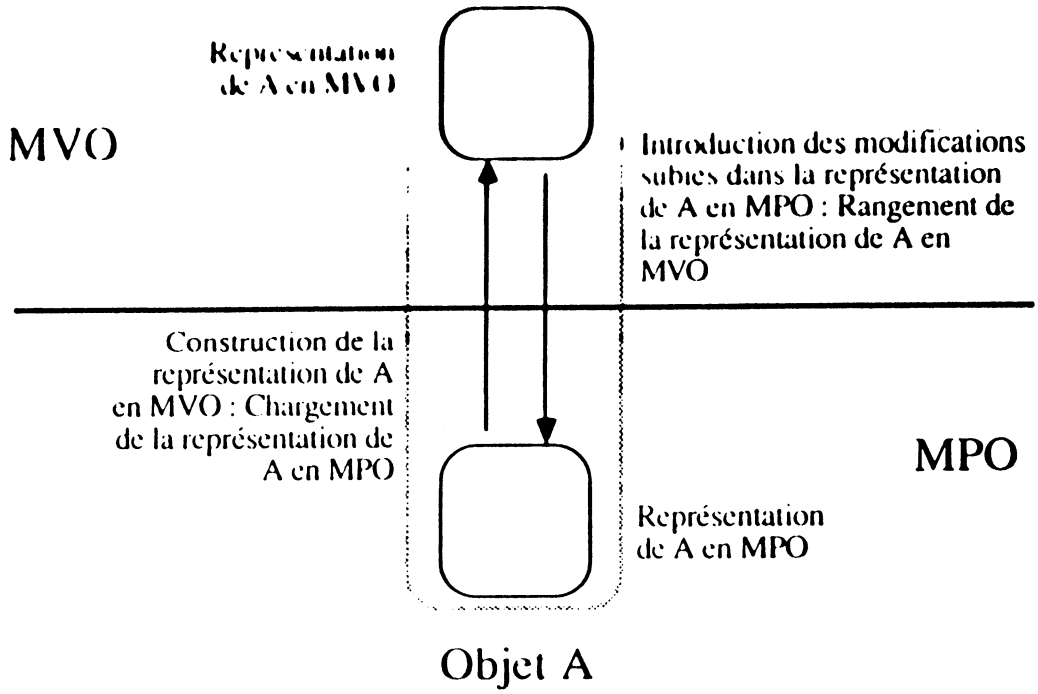


Figure 4.2 - L'image et l'objet stocké correspondants à un objet

L'union des mémoires virtuelles des processeurs virtuels de la machine à objets constitue la MVO du système Guide. En conséquence, elle est le support d'exécution des objets. La figure 4.3 montre les composants de la MVO et les situe par rapport aux autres éléments de la mémoire d'objets.

4.1.1.1 Images d'objet

Une image est matérialisée par le descripteur de l'objet en MVO et par un segment de mémoire virtuelle contenant l'état de l'objet. On appelle ce segment le **segment de l'image**. Un descripteur en MVO comporte les informations suivantes :

- L'oid de l'objet.
- La taille de l'objet.
- Des informations permettant de trouver le descripteur en MVO de la classe de l'objet.
- Le nombre de domaines ayant lié l'objet.
- L'identificateur du segment de l'image.

Un objet est **verrouillé** sur la mémoire virtuelle d'un processeur virtuel lorsqu'une image de l'objet ne peut être créée sur aucune autre des mémoires virtuelles qui composent la MVO. Un objet peut être verrouillé sur une mémoire virtuelle avant que

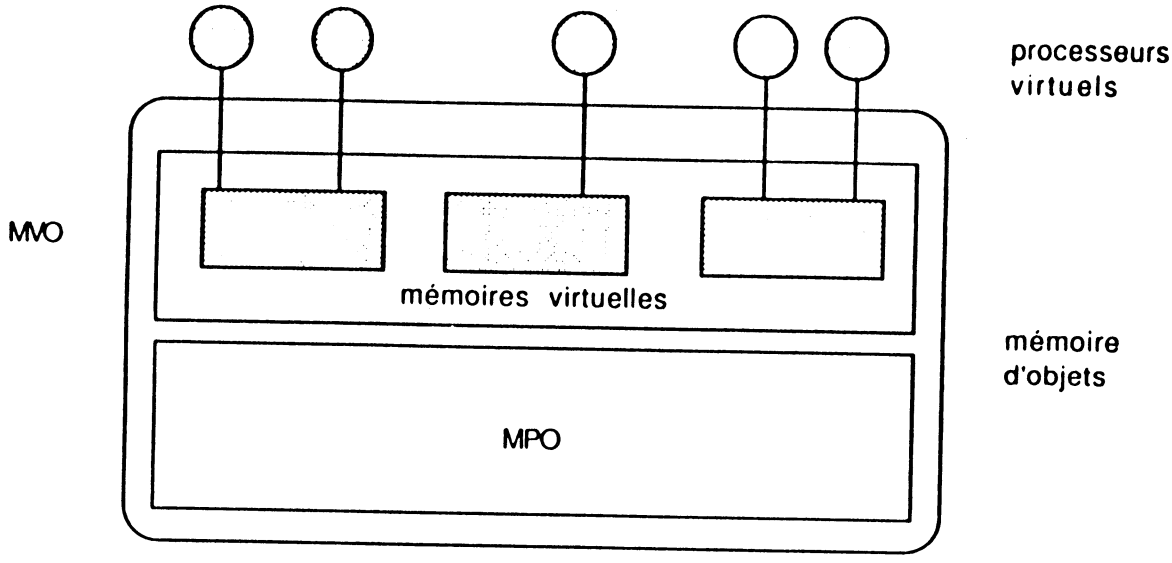


Figure 4.3 - Les composants de la MVO

son image ne soit créée sur cette mémoire virtuelle. Seulement lorsqu'il est explicitement **déverrouillé**, un objet peut à nouveau être verrouillé.

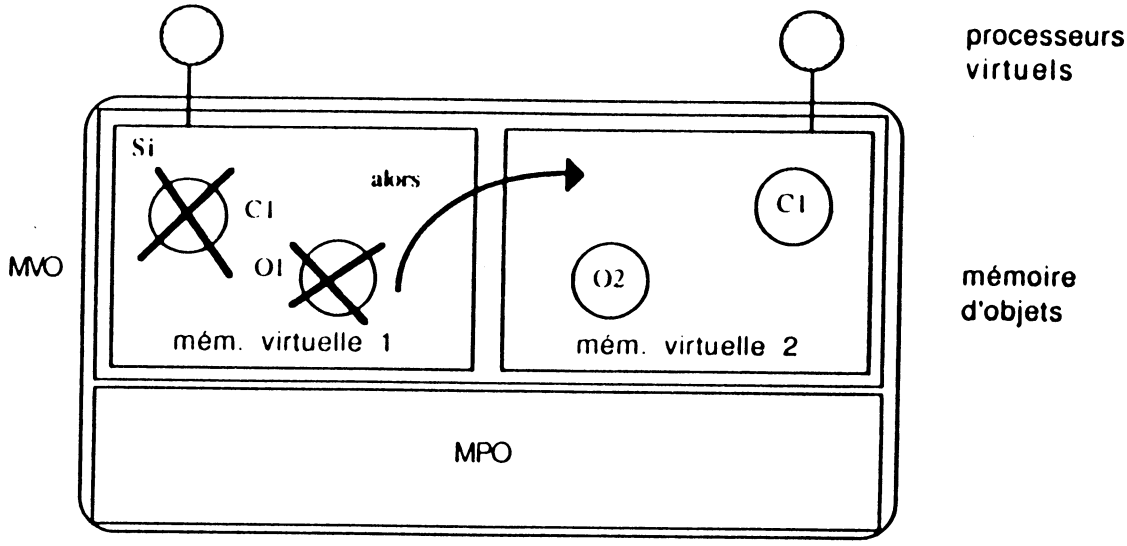
L'image d'un objet verrouillé doit être partagée par tous les domaines ayant lié l'objet, le verrouillage empêchant la création d'autres images. Le verrouillage et déverrouillage se trouvent donc à la base de la réalisation de l'un des principes de Guide qui est le partage d'objets.

Le mécanisme de verrouillage n'est cependant pas appliqué aux classes. En conséquence, l'image d'un objet peut être créée sur une mémoire virtuelle quelconque. La figure 4.4 illustre la raison pour laquelle une classe n'est pas verrouillée : L'objet C1 est une classe, il comporte le code des méthodes de ses instances. Si l'image de C1 ne pouvait pas être créée sur la mémoire virtuelle 1, l'image de O1 (instance de C1) serait forcément créée sur la mémoire virtuelle 2.

4.1.1.2 Création et destruction d'images

La création d'une image s'effectue en trois étapes :

1. Création du descripteur de l'objet en MVO. Ce descripteur est initialisé à partir des informations du descripteur en MPO (cf. section 4.1.2.2).
2. Allocation du segment de l'image.
3. Initialisation de ce segment. Elle se réalise en chargeant l'objet stocké correspondant. Si l'objet stocké lui-même n'a pas encore été initialisé (cf. section 4.1.2.3), le segment de l'image ne peut pas l'être. Il le sera plus tard, par l'exécution de la méthode `Init`, si celle-ci est définie sur l'objet.



O1 et O2 : Instances de C1
Si l'image de C1 ne pouvait être créée sur la mémoire virtuelle 1, l'image d'O1 serait forcée d'être créée sur la mémoire virtuelle 2 pour qu'on puisse exécuter le méthodes d'O1

Figure 4.4 - Les classes ne sont pas verrouillées

Une image ne peut être créée sur une mémoire virtuelle que si l'objet est verrouillé sur cette mémoire virtuelle. Il s'agit normalement de la mémoire virtuelle du processeur virtuel qui exécute l'appel d'une des méthodes de l'objet. Des critères d'optimisation internes permettront cependant de créer l'image sur une autre mémoire virtuelle [Lunati88].

La destruction d'une image consiste en la destruction du descripteur en MVO et en la libération de son segment de mémoire virtuelle. Une image ne peut être détruite que si l'objet correspondant est déverrouillé.

4.1.2 Description des objets en MPO

Afin de faciliter la fonction de localisation des objets stockés (cf. section 4.3.2) et aussi pour des raisons d'administration de ressources, on a choisi de ne pas banaliser complètement l'espace de stockage en mémoire secondaire. L'ensemble des objets stockés est donc décomposé en un certain nombre de sous-ensembles appelés **systèmes d'objets**[9].

9. Le terme 'système d'objets' a été choisi par analogie au terme 'système de fichiers' (*file system*). Cependant, son correspondant anglais est *container* plutôt que *object system*.

Un objet stocké n'est rangé que dans un seul système d'objets à la fois. Les objets stockés correspondants aux composants d'un objet *x* peuvent cependant être rangés dans des systèmes d'objets différents à celui de l'objet stocké correspondant à *x*. La composition d'un système d'objets évolue dynamiquement en fonction de la primitive `CreateObject` de la machine à objets.

L'union de tous les systèmes d'objets constitue la MPO du système Guide. En conséquence, elle est le support de conservation des objets. La figure 4.5 montre les composants de la MPO et les situe par rapport aux autres éléments de la mémoire.

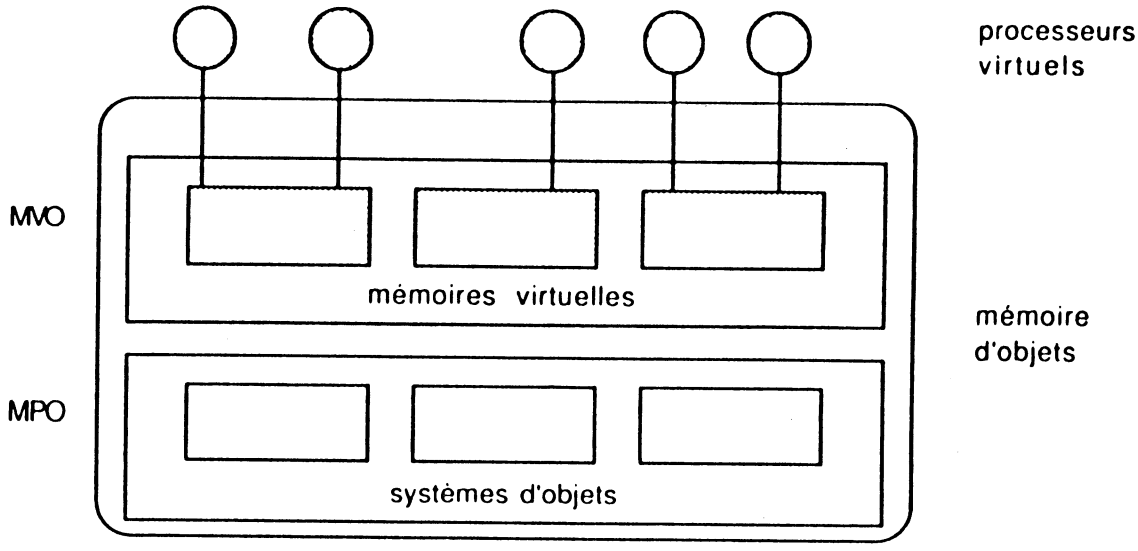


Figure 4.5 - Les composants de la MPO

4.1.2.1 Systèmes d'objets

Les raisons d'administration évoquées ci-dessus pour la décomposition de l'espace de stockage en systèmes d'objets appartiennent à deux catégories. Il y a d'abord les raisons de gestion de cet espace. Par exemple, un système d'objets ayant été alloué à une application, celle-ci est assurée de disposer de l'espace correspondant sans être perturbée par les applications dont les objets stockés sont rangés dans d'autres systèmes d'objets.

Les autres raisons sont celles du service d'administration des objets (cf. section 3.5). Un critère particulier d'administration peut être à l'origine de la définition d'un système d'objets. Parmi ces critères, on peut citer les suivants :

- Critère d'organisation. Tous les objets stockés créés par les membres d'une organisation (service, projet, etc.) sont rangés dans le même système d'objets.

- Critère d'application. Tous les objets stockés utilisés par une application sont rangés dans le même système d'objets.
- Critère de nature des objets. Par exemple, tous les objets stockés utilisés pour la gestion de types et classes sont rangés dans un système d'objets spécifique.
- Critère de performance. Le rangement des objets stockés est réparti entre plusieurs systèmes d'objets en fonction de l'utilisation et de la charge de chacun d'eux.

Ces différents critères peuvent bien sûr être combinés afin de permettre une gestion plus fine de l'espace de stockage. La définition d'un système d'objets sur la base d'un critère permet de plus l'adaptation des paramètres de gestion des systèmes d'objets (comme ceux de l'algorithme de vieillissement, cf. section 4.1.4) à la nature des objets stockés rangés sur chacun d'eux.

4.1.2.2 Objets stockés

Un objet stocké est matérialisé par le descripteur de l'objet en MPO et par un ensemble de blocs de système d'objets contenant l'état de l'objet. On appelle ces blocs les **blocs de l'objet stocké**. Un descripteur en MPO comporte les informations suivantes :

- L'oid de l'objet.
- La taille de l'objet.
- Des informations permettant de trouver le descripteur en MPO de la classe de l'objet.
- Les dates de création, de dernière lecture et de dernière modification.
- Des informations permettant de trouver les blocs de l'objet stocké.

En principe, un objet stocké est rangé dans un bloc de système d'objets. Cependant, si sa taille dépasse celle d'un bloc, il est rangé sous forme d'un arbre de blocs. Les objets de grande taille sont donc représentés en MPO par des objets stockés rangés dans un ensemble de blocs non contigus. Le descripteur en MPO ne comporte ainsi que l'adresse d'un bloc de système d'objets : celle du bloc qui contient soit tout l'état de l'objet, soit la racine de l'arbre de blocs.

Cette arborescence, inspirée des mécanismes mis en oeuvre dans le projet Exodus [Carey86], permet la réalisation des fonctions suivantes :

- L'adaptation du rangement des objets stockés à la fois aux objets de grande taille et à ceux de petite taille.
- L'extension dynamique de la taille des objets et l'insertion ou la suppression de données dans un bloc sans que les autres blocs de l'objet stocké soient modifiés.
- La validation (*commit*) d'une opération atomique de modification de l'état d'un objet par la seule écriture en mémoire stable du bloc racine de l'arbre. Ceci

concerne la réalisation des transactions [Ledot88].

4.1.2.3 Création d'objets stockés

La création d'un objet stocké s'effectue en trois étapes :

1. Création et initialisation du descripteur en MPO de l'objet.
2. Allocation des blocs de l'objet stocké.
3. Initialisation de ces blocs. Cette étape n'a lieu que lors du rangement du segment de la première image de l'objet.

Le système d'objets dans lequel un objet stocké est créé est déterminé en fonction d'un ensemble de critères d'administration. Par défaut, un objet stocké est rangé dans un système d'objets commun à tous les utilisateurs du système.

Un objet stocké ne disparait que lorsque l'objet correspondant est ramassé. La destruction d'un objet stocké consiste en la destruction du descripteur en MPO et en la libération de ses blocs.

4.1.3 Cycle de vie des objets

La création d'un objet peut maintenant être décrite en termes de la création de l'objet stocké et de la première de ses images. D'après le modèle de données, lorsqu'une activité exécute sur une classe la méthode `New`, une instance de cette classe est générée [Guide-R2] [Krakowiak88]. Dans le modèle de mémoire, cette génération consiste, d'une part, en la création et en l'initialisation du descripteur en MPO du nouvel objet et, de l'autre, en l'allocation des blocs de l'objet stocké.

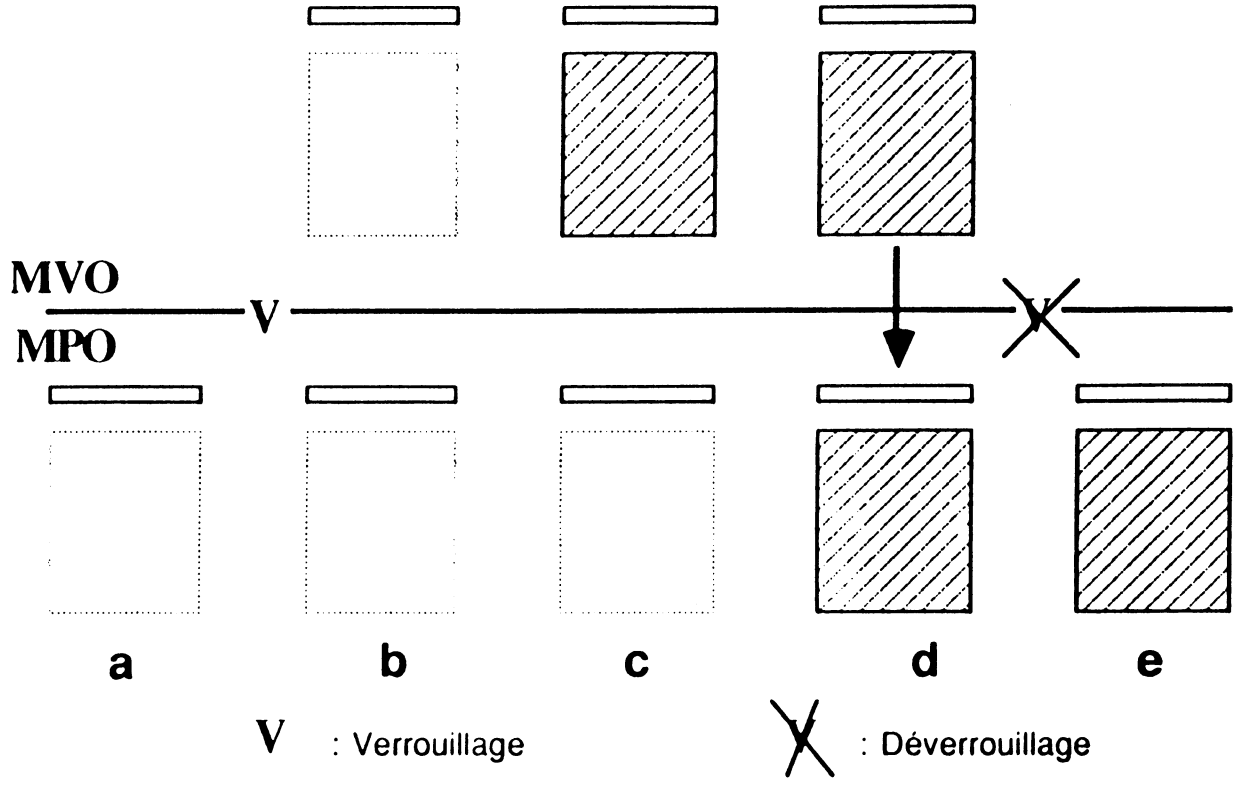
L'activité verrouille ensuite l'objet sur la mémoire virtuelle de son processeur virtuel d'exécution. La première image de l'objet est alors créée sur cette mémoire virtuelle. Puisque les blocs de l'objet stocké ne sont pas initialisés, le segment de l'image ne l'est pas non plus. L'objet est ensuite lié dans le domaine de l'activité.

En général, la méthode `Init` est la première qu'une activité exécute sur un objet. Que ce soit par celle-ci ou par une autre, le segment de l'image est initialisé. Enfin, l'image est rangée par la première fois en MPO et les blocs de l'objet stocké sont initialisés. Lorsque plus aucune méthode ne s'exécute, l'objet est délié du domaine et éventuellement déverrouillé. La figure 4.6 illustre la création d'un objet.

A partir de ce moment, l'objet mène une vie 'normale' : Les segments de ses images sont initialisés en chargeant les blocs de l'objet stocké. De même, les blocs de l'objet stocké sont modifiés comme conséquence du rangement du segment de l'image.

Le cycle de la vie d'un objet est détaillé par la suite.

Un objet est lié dynamiquement dans un domaine lorsqu'une activité de ce domaine appelle une méthode de cet objet et que celui-ci ne fait pas encore partie du domaine.



a. Création et initialisation du descripteur en MPO
Les blocs de l'objet stocké ne sont pas initialisés

b. Création et initialisation du descripteur en MVO
Le segment de l'image n'est pas initialisé

c. Initialisation du segment de l'image

d. Rangement en MPO de l'image ; initialisation des blocs de l'objet stocké

e. Destruction éventuelle de l'image

Figure 4.6 - Création d'un objet

L'objet ne peut cependant être lié que s'il est verrouillé sur une mémoire virtuelle et que s'il comporte une image sur cette mémoire virtuelle.

En conséquence, tout objet non verrouillé doit être verrouillé sur une mémoire virtuelle avant qu'un domaine puisse le lier. S'il n'a pas d'image sur cette mémoire virtuelle, une image doit être créée. On rappelle qu'une image ne peut être créée que si l'objet est verrouillé.

L'objet est délié lorsque plus aucune activité du domaine n'exécute de méthode sur lui. S'il n'est plus lié dans aucun domaine, il est alors déverrouillé. Cependant, il peut rester verrouillé dans une mémoire virtuelle. Si une de ses méthode est appelée, il doit être lié à nouveau.

Le segment de l'image est rangé en MPO seulement si l'objet est verrouillé. Le rangement du segment de l'image et le déverrouillage de l'objet sont deux fonctions indépendantes. Ceci rend possibles les rangements intermédiaires. Avant le déverrouillage de l'objet, le segment de son image doit cependant être rangé en MPO s'il a été modifié depuis son dernier rangement.

La figure 4.7 illustre les mécanismes de gestion de la mémoire d'objets mis en oeuvre pour lier et délier un objet d'un domaine.

On rappelle qu'une image ne peut être détruite que si l'objet est déverrouillé. Cependant, il est possible de déverrouiller un objet sans détruire son image. Ceci évite la création d'une nouvelle image lors d'une utilisation ultérieure. L'image conservée en MVO doit alors être validée avant toute nouvelle utilisation, l'objet ayant pu être verrouillé et sa nouvelle image modifiée entre temps.

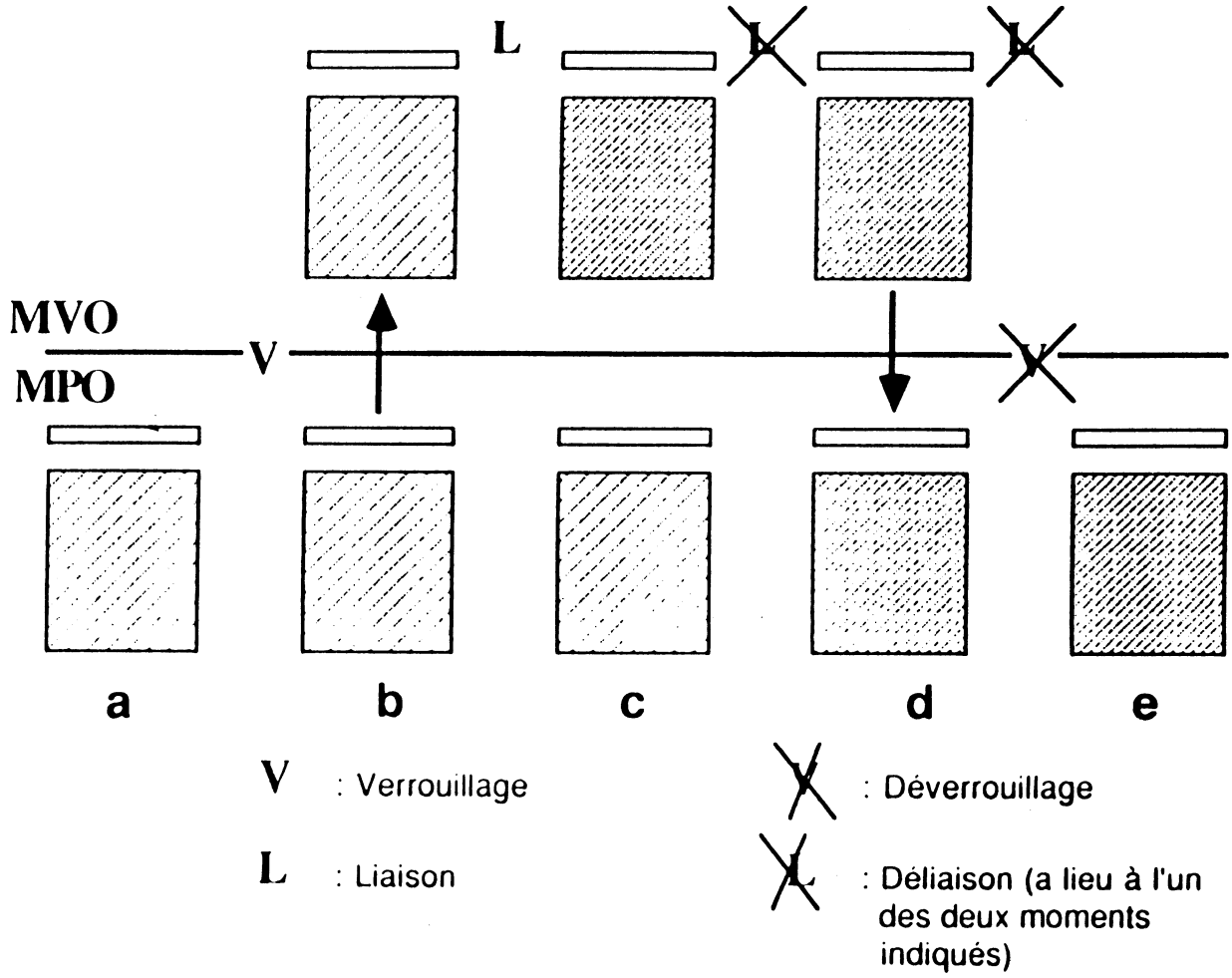
La figure 4.8 illustre les mécanismes de gestion de la mémoire d'objets mis en oeuvre pour la validation d'une image.

Finalement, le diagramme de transitions de la figure 4.9 explique le cycle de vie d'un objet en termes de son verrouillage, de son chargement en MVO et de sa liaison dans un domaine.

Il est intéressant de noter qu'au cours de sa vie, plusieurs images d'un objet peuvent être créées et qu'elles peuvent même exister simultanément, une seule étant cependant valide. L'idée de la figure 4.10 complète celle de la figure 4.2 : Le terme 'objet' correspond à plusieurs images mais seulement à un objet stocké.

Les fonctions de la gestion de la mémoire qui supportent le cycle de vie des objets sont les suivantes :

1. Création d'un objet stocké.
2. Verrouillage d'un objet.
3. Création d'une image.
4. Chargement en MVO des blocs d'un objet stocké (appelée aussi 'chargement d'un objet stocké').
5. Rangement en MPO du segment d'une image (appelée aussi 'rangement d'une image').
6. Déverrouillage d'un objet.
7. Validation d'une image.
8. Destruction d'une image.



a. L'objet ne comporte pas d'image

b. Après verrouillage, l'image est créée en chargeant en MVO l'objet stocké

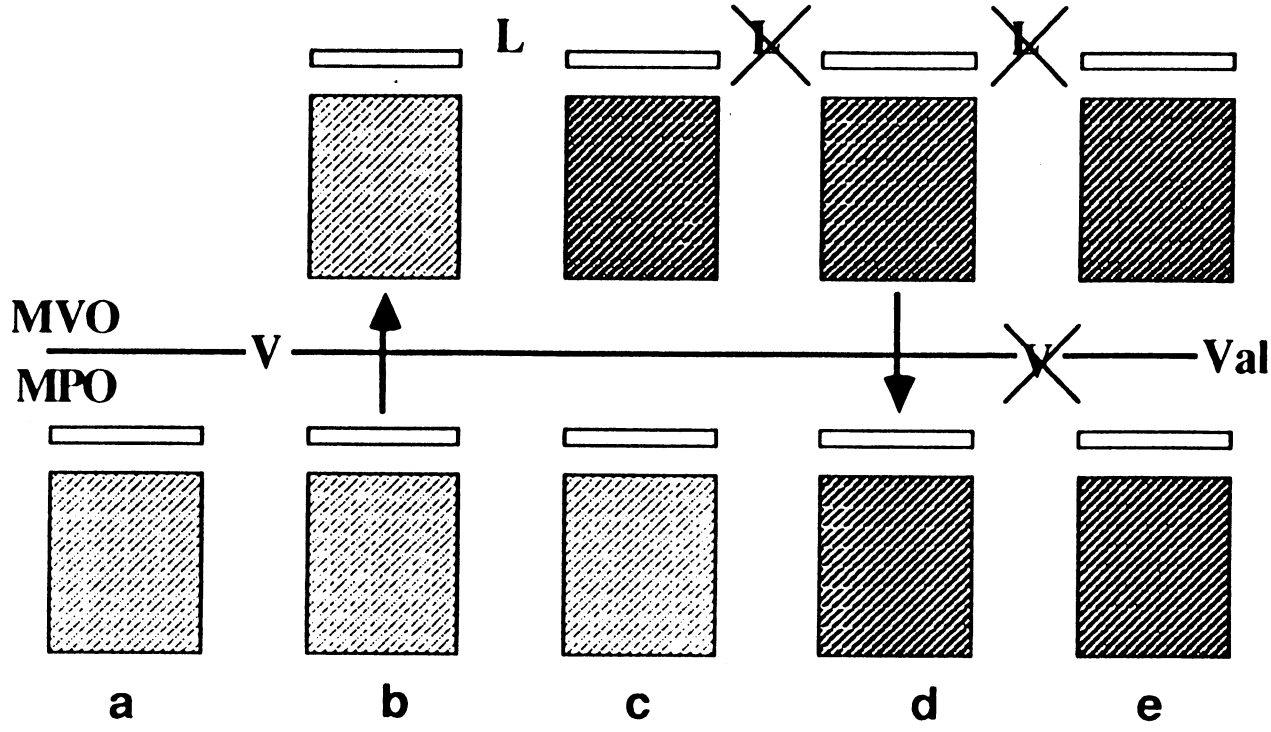
c. Les modifications se réalisent sur l'image lors de l'exécution des méthodes de l'objet

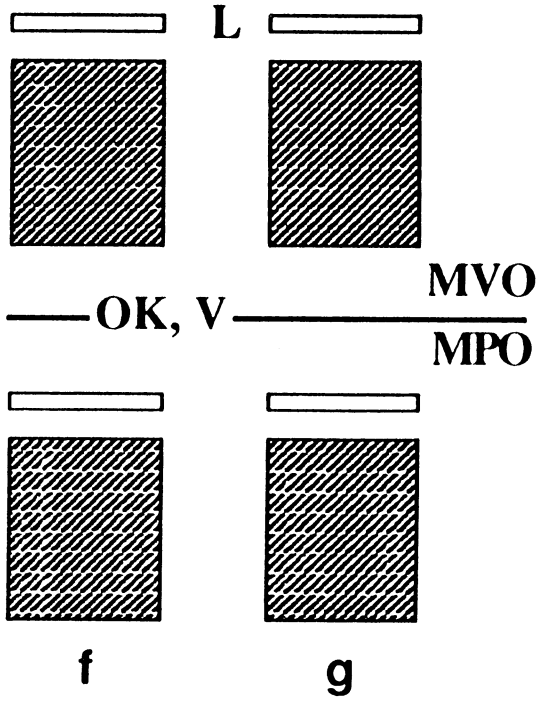
d. L'image est rangée en MPO

e. L'image est éventuellement détruite

Figure 4.7 - Support de la gestion de la mémoire d'objets à la liaison d'objets

9. Destruction d'un objet stocké.





L : Liaison

V : Verrouillage

Val : Demande de validation

OK : Validation acceptée (cf. chapitre 6 pour validation refusée)

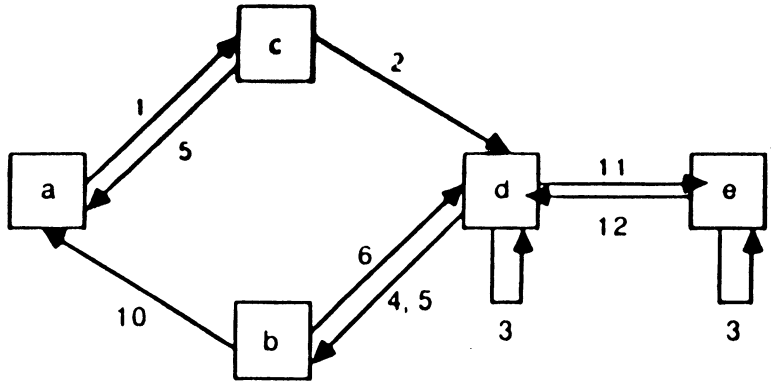
a, b, c, d. cf. figure 4.7

e. L'image est conservée même après le déverrouillage de l'objet

f. Réponse positive à la demande de validation

g. L'objet est à nouveau verrouillé puis lié

Figure 4.8 - Validation d'une image d'objet



Etats pour un objet

a : non lié, non verrouillé, non chargé

b : non lié, non verrouillé, chargé

c : non lié, verrouillé, non chargé

d : non lié, verrouillé, chargé

e : lié, verrouillé, chargé

Transitions

1. Verrouillage

2. Chargement

3. Rangement

4. Rangement + Déverrouillage

5. Déverrouillage

6. Validation acceptée

7, 8, 9. cf. chapitre 6

10. Destruction d'image

11. Liaison

12. Déliaison

Figure 4.9 - Cycle de vie d'un objet

Certaines fonctions ne concernent que la MVO ou que la MPO. D'autres constituent des véritables interactions entre les deux niveaux de la mémoire d'objets. Tel est le cas des fonctions de chargement et de rangement. La plupart de ces fonctions sont mises en oeuvre par les primitives de l'interface entre la MVO et la MPO. On verra au chapitre 6 la réalisation de cette interface.

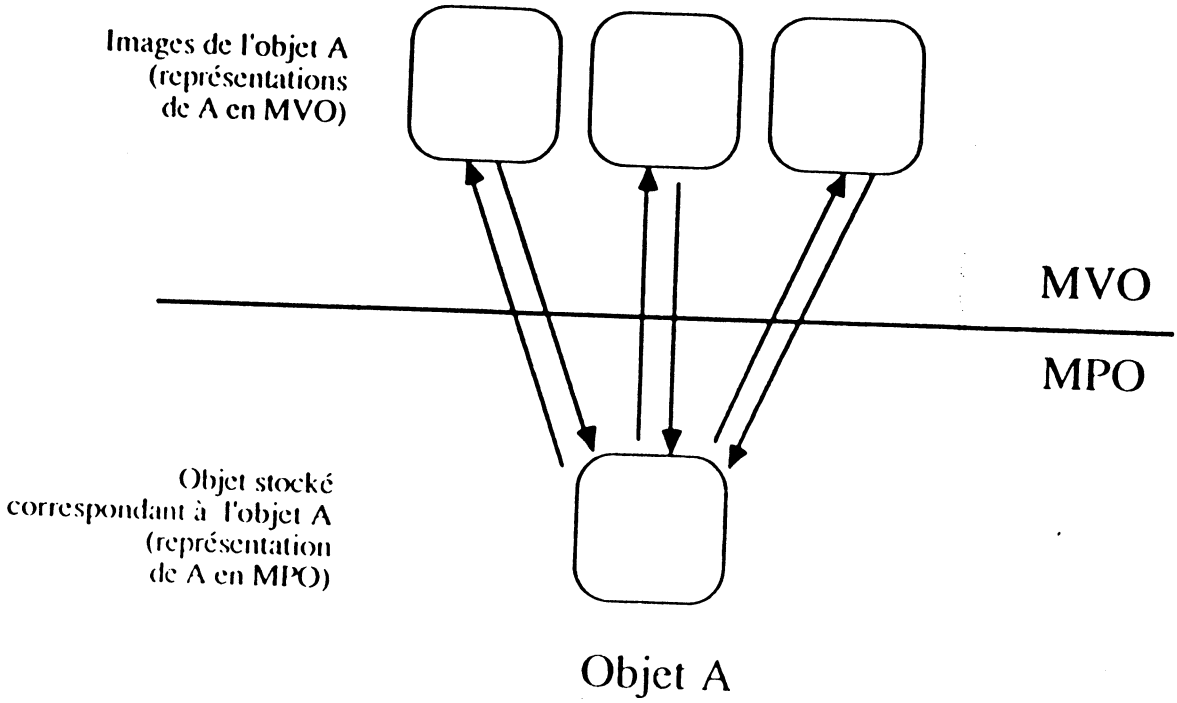


Figure 4.10 - Les images et l'objet stocké correspondants à un objet

4.1.4 Hiérarchie de la mémoire d'objets

La mémoire définie ci-dessus garantit aux objets une durée vie qui va au délà de la simple durée de l'exécution d'une activité. Elle est donc à la base de la réalisation de l'un des principes de Guide qui est la persistance des objets.

Un objet est ramassé lorsqu'il n'est plus accessible depuis la racine persistante. Cependant, même s'il est accessible depuis cette racine, il se peut qu'un objet soit peu utilisé, c'est-à-dire, que ses méthodes ne soient pas exécutées depuis un certain temps. Ces objets, qu'on appelle **vieux**, peuvent à terme encombrer la mémoire. Il est donc nécessaire de se donner un mécanisme pour les grouper et les isoler.

Le vieillissement des objets définit une organisation hiérarchique pour la mémoire. En effet, chaque système d'objets dépend d'un système d'objets **père** où sont rangés les objets stockés qui correspondent aux objets ayant vieilli. Plusieurs systèmes d'objets peuvent avoir le même père. Les systèmes d'objets pères sont eux-mêmes **fils** de systèmes d'objets où passent les objets stockés correspondant aux objets qui vieillissent encore.

Les systèmes d'objets sont donc organisés en arborescence. Le critère de vieillissement est un paramètre de gestion de chaque système d'objets. Il peut, par exemple, être fonction du taux d'occupation du système d'objets lui-même, ou encore de la nature des objets stockés qui y sont rangés. La figure 4.11 illustre le mécanisme

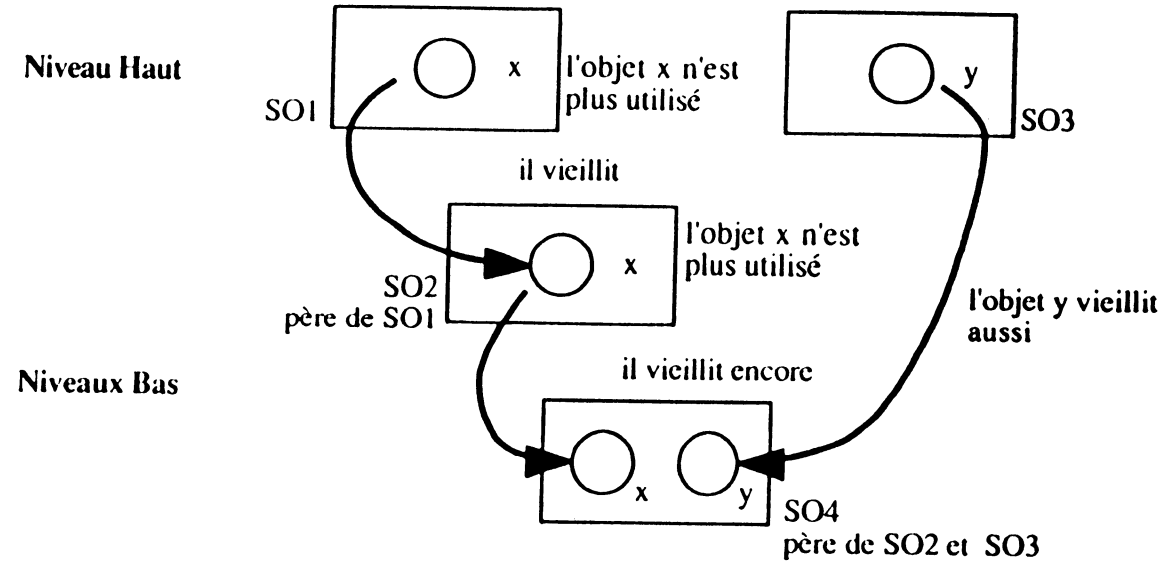


Figure 4.11 - Vieillissement des objets

de vieillissement.

Les méthodes d'un objet vieux ayant très peu de probabilités d'être exécutées, la MVO n'est en relation qu'avec les systèmes d'objets du niveau le plus haut de l'arborescence. Les objets stockés rangés dans les systèmes d'objets de ce niveau sont les seuls à être chargés en MVO pour l'initialisation des segments des images. De même, les images ne sont rangées en MPO que sur les blocs des objets stockés qui se trouvent dans les systèmes d'objets de ce niveau.

Lorsqu'une activité veut exécuter une méthode d'un objet vieux, celui-ci doit être d'abord **rajeuni**. Si l'objet stocké correspondant a redescendu plusieurs niveaux de l'arbre des systèmes d'objets, il est directement ramené dans l'un des systèmes d'objets de plus haut niveau. Les fonctions habituelles (verrouillage, chargement, liaison, etc.) sont ensuite réalisées pour l'exécution de la méthode. Après son rajeunissement, tout objet est de nouveau soumis au mécanisme de vieillissement normal.

La hiérarchie de la mémoire d'objets est illustrée par la figure 4.12. L'arborescence se compose de trois systèmes d'objets. L'objet stocké o1 passe du système d'objets A au système d'objets racine lorsque l'objet correspondant vieillit (2), et du système d'objets racine au système d'objets A lorsque l'objet correspondant rajeunit (1). Les blocs de l'objet stocké o2 sont chargés en MVO lorsqu'une des méthodes de l'objet correspondant va être exécutée (3) et, lorsque plus aucune de ses méthodes ne s'exécute, le segment de son image est rangé dans le système d'objets B (4).

On appelle **migration** d'un objet tout mouvement de ses images ou de l'objet stocké correspondant à l'intérieur de la mémoire. La plupart des migrations sont suscitées par les vieillissements et les rajeunissements. Sont aussi des migrations les mouvements internes à un niveau de la hiérarchie, la MVO y comprise.

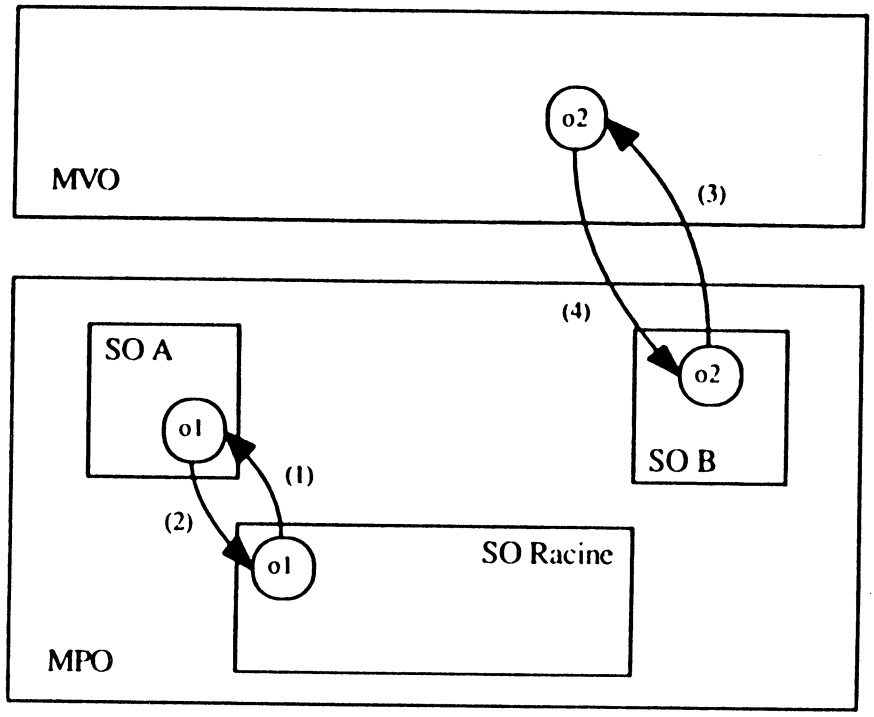


Figure 4.12 - La hiérarchie de la mémoire d'objets

Aucune raison de gestion de la mémoire ne motive la migration d'une image d'une mémoire virtuelle à une autre. En effet, les images sont partagées par les domaines indépendamment des mémoires virtuelles où elles sont présentes. En plus, le mécanisme de verrouillage ne s'appliquant pas aux classes, les images des objets peuvent être créées sur n'importe quelle mémoire virtuelle[10].

Les objets stockés rangés dans les systèmes d'objets autres que ceux du niveau le plus haut de l'arborescence ne migrent pas à l'intérieur de l'un de ces niveaux bas. Ils ne s'y trouvent qu'à cause du vieillissement des objets correspondants et le vieillissement implique, par définition, deux niveux différents.

---

10. Ce serait pour des raisons liées à la réalisation de Guide et non au modèle de la mémoire qu'on pourrait envisager les migrations en MVO. La répartion de charge et la récupération après pannes sont deux exemples de ces raisons. Or, la migration de l'image d'un objet serait souvent à l'origine d'une migration de l'activité qui exécute les méthodes de l'objet ou même du domaine qui a lié l'objet. La migration d'activités et domaines, d'une part, s'avère très difficile à réaliser et, de l'autre, ne fait même pas partie du modèle d'exécution [Guide-R3] [Guide-R5]. En conséquence, bien que faisable en théorie, la migration d'images n'a pas été retenue dans la réalisation du modèle de la mémoire de Guide (cf. chapitre 6 et Conclusions).

La migration à l'intérieur d'un niveau se réduit donc au cas où ce niveau est le plus haut dans l'arborescence des systèmes d'objets. Tandis que les vieillissements et les rajeunissements sont pris en compte automatiquement par les mécanismes décrits ci-dessus, les migrations des objets stockés à l'intérieur du niveau le plus haut se produisent par des raisons qui relèvent uniquement du service d'administration (cf. section 3.5).

Par exemple, dans le cas où un système d'objets est affecté à un organisme (laboratoire, université, entreprise, etc.), il doit être possible de faire migrer, d'un système d'objets à un autre, les objets stockés correspondant à tous les objets d'un utilisateur si cet utilisateur change d'organisme. Si parmi les objets de l'utilisateur se trouve un objet vieux, celui-ci est d'abord rajeuni, de la même façon que si une de ses méthodes allait être exécutée. La migration de l'objet stocké correspondant dans son nouveau système d'objets est ensuite effectuée.

Dorénavant, le terme 'migration d'un objet' fait référence uniquement à la migration de l'objet stocké correspondant à l'intérieur du niveau le plus haut de l'arborescence des systèmes d'objets.

La migration d'un objet constitue une fonction de la gestion de la mémoire d'objets, réalisée à la demande du service d'administration des objets. Pour des raisons de simplicité du modèle, aucune migration d'un objet stocké n'a lieu si l'objet correspondant comporte une image en MVO.

La migration d'objets est schématisée par la figure 4.13. L'objet stocké o1 migre du système d'objets A vers le système d'objets B (1). Si, lors de la demande de migration, l'objet stocké était dans le système d'objet racine, l'objet correspondant serait au préalable rajeuni et o1 serait rangé dans le système d'objets A (2).

4.2 Désignation d'objets

Une référence permet de désigner un objet. Les références sont utilisées et interprétées à différents niveaux du système :

- Les **références-système** sont interprétées par la gestion de la MPO.
- Les **références d'exécution** sont interprétées par la gestion de la MVO.
- Les **références-langage** sont interprétées par la gestion de domaines et d'activités.

N'étant manipulées que par les mécanismes d'exécution, les références-langage ne sont pas concernées par la gestion de la mémoire. Leur définition et utilisation sont décrites dans [Guide-R2] et [Guide-R5].

Le format des références d'exécution et des références-système est illustré par la figure 4.14.

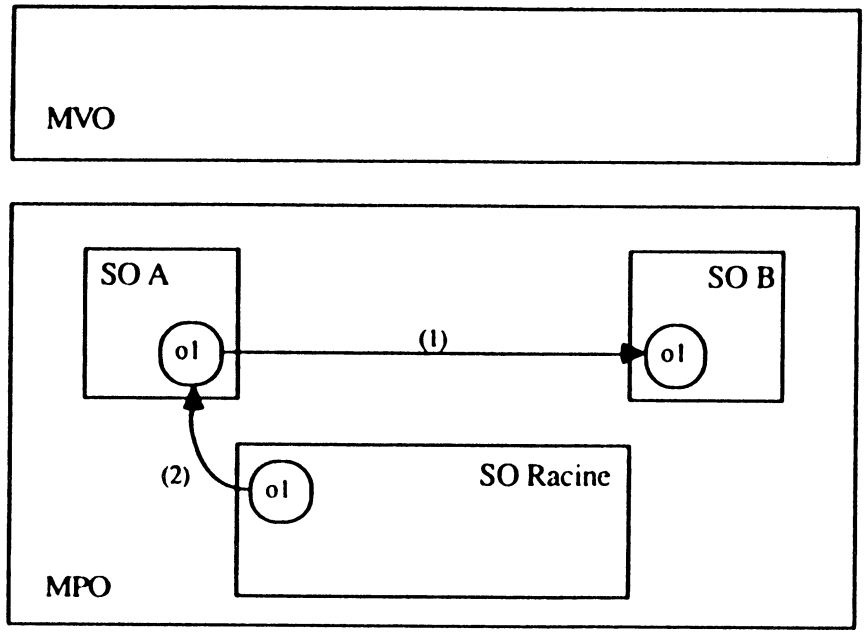


Figure 4.13 - Migration d'objets

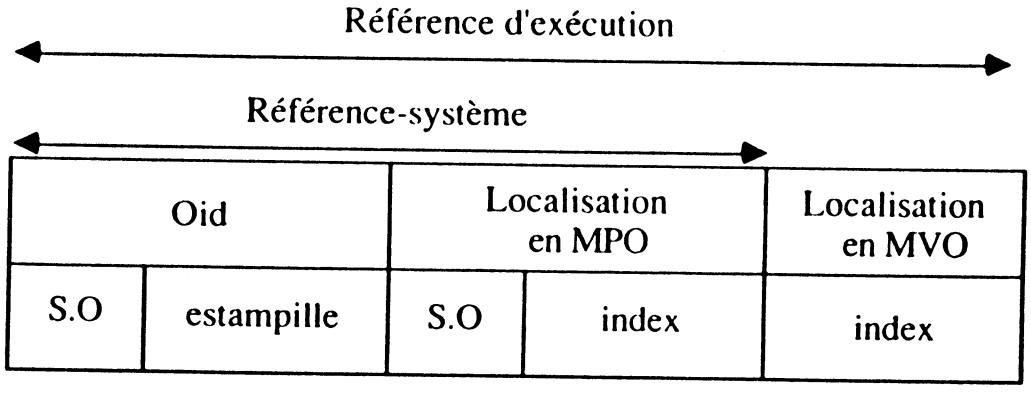


Figure 4.14 - Références d'exécution et -système

Références-système

Une référence-système est composée de l'oid de l'objet et d'un indicateur de localisation en MPO.

L'oid d'un objet se compose de :

- L'identificateur du système d'objets où l'objet stocké correspondant est rangé au moment de sa création. Ce **système d'objets de création** appartient au niveau le

plus haut de l'arborescence des systèmes d'objets.

- Une estampille qui est le numéro d'ordre l'objet dans son système d'objets de création.

L'indicateur de localisation en MPO d'un objet se compose de :

- L'identificateur du système d'objets où l'objet stocké correspondant est rangé couramment. Ce **système d'objets de résidence** appartient au niveau le plus haut de l'arborescence des systèmes d'objets. Il est différent du système d'objets de création si l'objet stocké a migré.
- L'index de l'objet stocké correspondant dans le système d'objets de résidence. Le terme 'index' est utilisé pour désigner un mécanisme d'accès direct (adresse d'un bloc) ou indirect (position dans une table) à l'objet stocké.

La première référence-système d'un objet est établie à la suite de la création de l'objet stocké correspondant. Ceci présente les avantages suivantes :

- La référence-système est toujours valide, même avant que le segment de l'image ne soit rangé par la première fois en MPO.
- Les segments des images des objets qui contiennent une référence sur un objet x peuvent être rangés en MPO *avant* celui de x.
- Il est impossible de créer un objet dont le segment de l'image correspondante ne puisse pas être rangé en MPO par la suite.

Après la première, toutes les autres références-système d'un objet sont initialisées par copie d'une déjà existante.

Lorsqu'un objet migre ou vieillit, l'indicateur de localisation que comportent ses références-système devient caduc. Puisqu'il est impossible de connaître la totalité des objets qui maintiennent des références-système sur un objet x[11], ces références-système ne peuvent pas être mises à jour lorsque x vieillit ou migre. Les références-système sont donc mises à jour une à une, au fur et à mesure qu'elles sont utilisées (cf. section 4.3.2).

Les objets étant persistants, leurs références-système ne sont pas détruites.

Références d'exécution

Une référence d'exécution est composée d'une référence-système de l'objet et d'un indicateur de localisation en MVO.

---

11. Ceci est faisable mais très coûteux à mettre en oeuvre.

L'indicateur de localisation en MVO d'un objet est la position qu'occupe son descripteur en MVO à l'intérieur de la mémoire virtuelle où son image a été créée. L'utilisation d'une référence d'exécution est donc restreinte à une mémoire virtuelle.

La référence d'exécution d'un objet est établie lorsque l'image de celui-ci est créée sur une mémoire virtuelle. A la suite d'une réorganisation interne de cette mémoire virtuelle, l'indicateur de localisation peut devenir caduc. La référence d'exécution est mise à jour seulement lorsqu'elle est utilisée pour la localisation de l'image (cf. section 4.3.1).

Une référence d'exécution est détruite dès que l'image de l'objet est supprimée.

4.3 Localisation d'objets

La localisation de l'image d'un objet a lieu lorsqu'une activité a besoin de verrouiller cet objet sur une mémoire virtuelle. Ceci afin de le lier dans son domaine et de pouvoir exécuter ensuite l'une de ses méthodes.

Si la recherche de l'image a été infructueuse, c'est-à-dire, si l'objet ne comporte pas de représentation en MVO, l'objet stocké correspondant doit être localisé afin que le chargement de ses blocs permette la création de l'image.

Le support de la localisation d'objets

Deux cas particuliers peuvent se présenter lors de la recherche de l'image d'un objet sur une mémoire virtuelle :

1. L'image sur cette mémoire virtuelle n'est pas valide.
2. L'objet est verrouillé sur une autre mémoire virtuelle.

La gestion d'une mémoire virtuelle n'a connaissance ni des objets verrouillés sur d'autres mémoires virtuelles ni de ceux non verrouillés en MVO. Ceci évite d'avoir à maintenir une cohérence entre les différentes mémoires virtuelles, mais implique la nécessité d'avoir recours à une fonction qui indique si un objet a été verrouillé ou non. A partir d'une référence-système, cette fonction, dite **de verrouillage**, permet à tout instant de savoir si l'objet est verrouillé ou non, et, si oui, sur quelle mémoire virtuelle.

De toute évidence, la gestion d'une mémoire virtuelle n'est pas non plus concernée par les modifications subies par les objets verrouillés ailleurs. Le souci de validation d'images impose cependant le besoin de déterminer, à l'aide d'une autre fonction, si un objet a été modifié ou non. A partir d'une référence-système, cette fonction, dite **de validation**, permet de savoir si l'objet à été modifié ou non depuis son dernier déverrouillage.

Une fois déterminée la mémoire virtuelle où se trouve l'image d'un objet, il est indispensable de localiser à l'intérieur son descripteur en MVO. Si l'indicateur de localisation en MVO de la référence d'exécution est caduc, un mécanisme de parcours

de tous les descripteurs en MVO présents dans cette mémoire virtuelle est nécessaire.

Deux cas particuliers peuvent se présenter lors de la recherche d'un objet stocké dans son système d'objets de résidence :

1. L'objet a vieilli.
2. L'objet stocké a migré.

La gestion d'un système d'objets n'ayant pas connaissance des objets stockés rangés dans d'autres systèmes d'objets, il est nécessaire de prévoir un mécanisme qui permette de retrouver un objet stocké dont la localisation en MPO a changé.

Dans le cas du vieillissement, ce mécanisme consiste à redescendre, de fils en père, les niveaux bas de l'arborescence des systèmes d'objets jusqu'à retrouver l'objet stocké. Pour les migrations, le mécanisme se réduit à l'appel d'une fonction qui indique si l'objet stocké a effectivement migré ou non. A partir d'une référence-système, cette fonction, dite **de migration**, permet de savoir si un objet stocké a migré ou non et, si oui, sur quel système d'objets.

Après la détermination du système d'objets de résidence d'un objet, il est essentiel de localiser à l'intérieur son descripteur en MPO. Si l'indicateur de localisation en MPO de la référence-système ne suffit plus, un mécanisme de parcours de tous les descripteurs en MPO présents dans ce système d'objets est nécessaire[12].

4.3.1 Localisation d'images

La recherche de l'image d'un objet a lieu à partir de la mémoire virtuelle d'exécution de l'activité qui veut verrouiller cet objet. On appelle cette mémoire la mémoire virtuelle **courante**.

La recherche s'effectue en utilisant l'indicateur de localisation en MVO contenu dans la référence d'exécution. Cet indicateur permet de récupérer le descripteur de l'objet en MVO. Si l'oid contenu dans le descripteur est égal à l'oid contenu dans la référence, l'image recherchée est effectivemet trouvée dans la mémoire virtuelle courante.

Le fait que ces deux oid's soient différents montre que l'indicateur de localisation en MVO n'est plus valide. Dans ce cas, l'image doit être recherchée en utilisant le mécanisme de parcours des descripteurs en MVO. Si l'image est trouvée, l'indicateur de localisation en MVO est mis à jour dans la référence d'exécution. Ceci évite une nouvelle recherche lors de la prochaine utilisation de cette référence.

---

12. Des outils performants de recherche, comme le *hash coding* à partir des oids, ont été mis en oeuvre dans l'implantation de la localisation des objets. Voir chapitre 6.

Si dans la mémoire virtuelle courante se trouve une image ayant besoin de validation, la fonction de verrouillage est utilisée pour déterminer si l'objet correspondant a été verrouillé sur une autre mémoire virtuelle. Si c'est le cas, l'image dans la mémoire virtuelle courante doit être abandonnée au profit de celle qui se trouve (ou qui va bientôt être créée) sur l'autre. Une nouvelle référence d'exécution est créée pour l'objet sur cette mémoire virtuelle. L'indicateur de localisation en MVO est déterminé en utilisant le mécanisme de parcours des descripteurs en MVO.

Si l'objet correspondant à l'image qu'on valide n'est pas verrouillé, la fonction de validation est utilisée pour déterminer si cet objet a subi des modifications. Si c'est le cas, l'image doit être abandonnée et par la suite tout se passe comme si elle n'existait pas en MVO. Si ce n'est pas le cas, l'image est considérée comme valide.

Si l'image qu'on cherche ne se trouve pas sur la mémoire virtuelle courante, la fonction de verrouillage est appelée. Si l'objet correspondant n'est pas verrouillé dans une autre mémoire virtuelle, l'image qu'on essaie de localiser n'existe pas en MVO. Dans le cas contraire, l'image se trouve (ou va bientôt être créée) sur une autre mémoire virtuelle. Comme avant, une nouvelle référence d'exécution, valide sur cette autre mémoire virtuelle, est créée pour l'objet.

Si l'image qu'on cherche n'existe pas en MVO, elle doit être créée en chargeant l'objet stocké correspondant. On appelle cette situation un **défaut d'objet**.

4.3.2 Localisation d'objets stockés

La recherche d'un objet stocké s'effectue en utilisant l'indicateur de localisation en MPO contenu dans la référence-système. Elle a donc lieu à partir du système d'objets de résidence de l'objet correspondant.

L'indicateur de localisation en MPO permet de récupérer le descripteur de l'objet en MPO. Si l'oid contenu dans le descripteur est égal à l'oid contenu dans la référence, l'objet stocké recherché est effectivemet trouvé dans le système d'objets de résidence.

Le fait que ces deux oid's soient différents montre que l'objet stocké ne se trouve pas dans le système d'objets de résidence et que, par conséquence, l'indicateur de localisation en MPO n'est plus valide. La fonction de migration est utilisée pour déterminer si l'objet stocké a migré. Si c'est le cas, l'objet stocké doit être recherché dans le nouveau système d'objets. Ceci est réalisé en utilisant le mécanisme de parcours des descripteurs en MPO. L'indicateur de localisation en MPO est ensuite mis à jour dans la référence-système. On évite ainsi un nouvel appel à la fonction de migration et une nouvelle recherche lors de la prochaine utilisation de cette référence.

On considère maintenant le cas où l'objet stocké qu'on recherche n'a pas migré. L'objet correspondant ayant pu vieillir, il faut redescendre l'arborescence de systèmes d'objets. L'objet stocké doit être recherché dans tous les systèmes d'objets visités au cours de cette descente. Ceci est réalisé en utilisant, sur chaque système d'objets, le mécanisme de parcours des descripteurs en MPO.

Si l'objet stocké est trouvé dans l'un de ces systèmes d'objet, l'objet correspondant est rajeuni. Ceci est réalisé en rangeant l'objet stocké dans le système d'objets désigné par l'indicateur de localisation en MPO de la référence-système.

Si, après la descente de l'arborescence jusqu'à la racine, l'objet stocké n'est toujours pas trouvé, il n'existe pas en MPO. Soit il a été détruit, soit la référence-système comporte des informations erronées.

4.4 Récupération d'espace

La mémoire de Guide comporte uniquement des objets persitants. Par conséquent, qu'il soit miette ou non, tout objet possède un objet stocké. Ainsi, une fois son image rangée en MPO et elle-même déverrouillée, une miette peut être décelée en MPO et son espace récupéré.

Contrairement à beaucoup de systèmes à objets, où la récupération de l'espace des miettes est faite en mémoire principale [Decouchant87], dans Guide, cette récupération peut être effectuée sur leurs objets stockés correspondants. Dans ce cas, la récupération de l'espace que l'image d'une miette occupe sur une mémoire virtuelle a lieu de la manière habituelle, c'est-à-dire, lorsque la miette est déverrouillé et que la gestion de cette mémoire virtuelle a besoin de cet espace pour la création d'une autre image.

Rien n'empêche cependant qu'une ou plusieurs miettes puissent être détectées et ramassées en MVO plutôt qu'en MPO. Pourvu que leurs objets stockés correspondants soient détruits, cette manière de récupération d'espace est tout à fait valable. Etant donné que la réalisation d'un ramasse-miettes ne rentrait pas dans le cadre des objectifs de la première étape du projet Guide, les concepteurs de ce système ne se sont pas encore décidés par un ramassage purement en MPO ou un ramassage sur l'espace global d'objets. Les deux solutions qu'on présente par la suite restent donc à l'étude. De toute manière, chacune s'adapte parfaitement aux conditions requises par le modèle de la mémoire persistante de Thatte.

4.4.1 Ramasse-miettes sur l'espace global d'objets

Le ramassage de miettes peut se faire selon l'un des deux algorithmes suivants : Algorithme de marquage de Dijkstra [Dijkstra78] et algorithme de génération d'Ungar [Ungar84]. Une première proposition d'adaptation de ces algorithmes à la mémoire de Guide se trouve dans [Nguyen Van89]. On ne présente ici que les points les plus importants.

Dans l'algorithme de marquage, la fermeture transitive de la racine persistante est parcourue et les objets visités sont tous marqués. Les objets non marqués sont ensuite ramassés car ils sont des miettes. Les images d'une miette, si elle en a, et l'objet stocké correspondant sont détruits.

La récupération de l'espace d'un cycle de miettes se complique lorsque les images et objets stockés correspondants à ces miettes se trouvent dans des mémoires virtuelles

différentes ou dans des systèmes d'objets différents. Une phase de synchronisation dans le marquage, basée sur le même principe que l'algorithme de ramasse-miettes de la mémoire d'objets répartie de Decouchant (cf. section 2.2.1.2.4), doit être mise en oeuvre pour détecter ces cycles.

Cet algorithme peut être exécuté en parallèle avec des activités Guide pourvu que celles-ci respectent certaines règles comme, par exemple, ne pas marquer une miette et ne pas ôter la marque d'un objet.

L'idée de base du deuxième algorithme résulte des observations faites par Ungar sur l'ensemble des objets d'un système Smalltalk. Selon Ungar, un objet Smalltalk qui reste en mémoire principale au-délà d'un certain seuil de temps a très peu de probabilités de devenir une miette.

L'espace des objets est ainsi divisé en deux : les objets 'durables' et les objets 'éphémères'. Les premiers ont atteint le seuil requis de temps, ils ne seront donc pas ramassés. Une liste regroupe les objets durables qui possèdent au moins une référence vers un objet éphémère. Cette liste constitue en fait la racine de persistance.

Au bout d'un certain temps, quand l'age des objets éphémères augmente, le ramassage a lieu. On analyse les éphémères pointés par les durables de la liste : ceux qui dépassent le seuil deviennent durables et les autres sont marqués. Ensuite, ces mêmes actions se repètent pour les objets qui appartiennent aux fermetures transitives des objets qu'on vient de rendre durables et de ceux qu'on vient de marquer. Les objets qui, après ces actions, ne sont ni durables ni ont été marqués sont des miettes. Leurs images et objets stockés sont détruits. Les objets marqués sont ensuite démarqués.

Cet algorithme ne peut s'exécuter en parallèle avec aucune activité.

4.4.2 Ramasse-miettes en MPO seulement

La récupération de l'espace de l'image d'un objet sur une mémoire virtuelle a lieu lorsque l'objet est déverrouillé et que la gestion de cette mémoire virtuelle a besoin de cet espace pour la création d'une autre image. Cette récupération se réduit donc à la reallocation des descripteurs en MVO et des segments des images des objets déverrouillés. Le vrai ramasse-miettes a lieu en MPO.

La récupération de l'espace qu'occupe en MPO un groupe de miettes se complique lorsque les objets stockés correspondants à ces miettes sont rangés dans des systèmes d'objets différents. Afin de la rendre plus facile, on restreint la récupération d'espace aux objets stockés rangés dans la racine de la hiérarchie de la mémoire d'objets. Puisque leurs méthodes ne peuvent pas être exécutées, les miettes vieillissent plus rapidement que les autres objets et, en plus, elles ne sont pas soumises au rajeunissement. En conséquence, après un certain temps, tous les membres d'un groupe de miettes se trouvent dans la racine.

Le ramassage de miettes a donc lieu dans la racine de la hiérarchie de la mémoire d'objets. Un algorithme de marquage est lancé depuis la racine de persistance mais sont

pris seulement en compte les objets de la racine de la hiérarchie de la mémoire. Les objets stockés qui n'ont pas été marqués sont détruits.

4.5 Duplication d'objets

Un objet est **dupliqué** lorsqu'il existe en plusieurs exemplaires sur la mémoire d'objets. La duplication permet donc d'augmenter la disponibilité des objets mais rend plus difficile leur gestion à cause du maintien de la cohérence entre exemplaires. La gestion de la mémoire d'objets ne considère pas les exemplaires d'un objet comme des objets différents. En fait, à partir de la référence-système d'un objet, il doit être possible de localiser tous ses exemplaires.

Un objet est dupliqué s'il possède plusieurs images, plusieurs objets stockés ou les deux. En fait, la seule duplication permise en MVO est celle qu'on a décrite précédemment : un objet peut avoir plusieurs images mais une seulement est valide. Les classes sont cependant des exceptions à cette règle, ce qui fait d'elles les seuls objets vraiment dupliqués en MVO.

La duplication en MPO résulte de l'organisation interne des systèmes d'objets. Un système d'objets se compose d'un ensemble de **volumes** identiques. Un volume est une suite de blocs. En conséquence, les objets stockés rangés dans un système d'objets sont tous dupliqués dans chaque volume et une fois par volume. Ceci évite d'avoir à maintenir, pour chaque copie d'objet stocké sur un volume, la liste des autres volumes sur lesquels existe une copie.

Un système d'objets comporte un volume **principal**, où les images sont rangées après modification, et des volumes **secondaires**, où ces modifications sont propagées. Seulement dans le cas de l'image d'un objet atomique, la propagation des mises à jour dans les volumes secondaires du système d'objets est effectuée de façon synchrone. Les mécanismes de gestion de la duplication sont inspirés de ceux mis en oeuvre dans le système Locus [Popek85].

Le nombre de copies des objets stockés rangés dans un système d'objets relève de la gestion de la mémoire. Des décisions sont prises en fonction du degré de disponibilité souhaité pour les objets ainsi que de la disponibilité et fiabilité des supports physiques. Ceci rend possible la modification du degré de duplication des objets stockés sans avoir à modifier le code des applications qui les ont créés.

5. Désignation symbolique et gestion de versions

En plus des fonctions de gestion de la mémoire, la gestion d'objets de Guide réalise les fonctions de désignation symbolique d'objets et de gestion des versions des objets. Il s'agit des fonctions qui mettent en oeuvre la désignation par le biais de noms symboliques et la création et la manipulation de versions.

Ces fonctions ont été spécifiées et implantées, pour des raisons déjà mentionées (cf. section 3.5), comme des services du système Guide.

Les fonctions du service de désignation symbolique (aussi dit 'service de noms') sont mises en oeuvre par les méthodes d'un ensemble d'objets et non pas par un domaine serveur (cf. section 3.5). Il en est de même pour les fonctions du service de versions. Chacun de ces deux ensembles est constitué par les instances d'une classe particulière, définie en langage Guide et décrite ultérieurement (cf. chapitre 7).

5.1 Le service de désignation symbolique

Les références permettent de manipuler les objets soit de manière interne à la machine virtuelle (références d'exécution et -système), soit à l'intérieur du code d'une classe (références-langage). Il y a cependant des cas où les références ne constituent pas un moyen confortable de manipulation d'objets.

Prenons l'exemple du lancement d'une application de façon interactive. Chaque fois qu'un usager doit le faire, il est censé se souvenir non seulement de la référence de l'objet initial du domaine correspondant à l'application mais aussi de celles des paramètres de la méthode initiale. Or, les références sont des suites d'octets qui n'ont aucun sens pour un usager final.

C'est pour cela qu'une nouvelle sorte de désignation, la désignation par noms symboliques, est introduite. Un nom symbolique est une suite de caractères que les usagers eux-mêmes peuvent définir. Ainsi, des noms symboliques sont donnés aux objets, afin de les désigner aisément.

Le principe de la désignation symbolique est le suivant : Le nom symbolique d'un objet doit permettre de retrouver une référence-système sur celui-ci. Une fois la référence en main, toutes les manipulations sur l'objet sont possibles. Seuls les objets persistants peuvent avoir un nom.

Le service de désignation symbolique se charge de mettre en oeuvre ce type de désignation. Ses fonctions de base sont donc les suivantes :

- Associer le nom symbolique (ou les noms symboliques) d'un objet à une référence de cet objet.
- Retrouver la référence d'un objet à partir du nom symbolique (de l'un des noms symboliques) de celui-ci.

- Dissocier la référence d'un objet du nom symbolique (de l'un des noms symboliques) de celui-ci.

Du point de vue d'un usager, l'association d'un nom symbolique à une référence consiste simplement à donner un nom à un objet.

Le service de désignation symbolique se charge en outre de la gestion des noms symboliques eux-mêmes. Les fonctions de base de cette gestion sont les suivantes :

- Introduction, dans l'espace des noms symboliques, d'un nom défini par un usager.
- Suppression d'un nom symbolique de cet espace.

5.1.1 L'espace de noms

L'espace des noms symboliques est hiérarchique. On a préféré cette solution, contre celle d'un espace plat, parce que l'expérience des systèmes de gestion de fichiers (cf. chapitre 1) enseigne que la gestion d'un grand nombre de noms est plus efficace si l'espace de ces noms est hiérarchique. Or, les tendances d'utilisation de certaines caractéristiques du système Guide (la composition d'objets et la programmation de classes dont les instances sont de taille petite ou dépourvues d'objets internes) permettent de prévoir que le nombre d'objets manipulés sera grand.

La construction de l'espace de noms se réalise de manière classique : Pour pouvoir les gérer, les noms symboliques d'objet sont groupés. On appelle **répertoire** un groupe de noms symboliques. Chaque répertoire possède aussi un nom symbolique qui permet de désigner le groupe de noms qu'il comporte. La hiérachie est établie dès qu'on autorise les répertoires à grouper non seulement des noms d'objet mais aussi des noms de répertoire. Il y a même des répertoires qui ne comptent que des noms de répertoire. La figure 5.1 montre la hiérarchie de noms.

Dans le graphe de la figure 5.1, les noeuds terminaux correspondent à des noms d'objet et les noeuds non terminaux à des noms de répertoire. Désormais, le terme 'nom symbolique' s'applique aussi bien aux noms d'objet qu'aux noms de répertoire.

Une fois définie la structure de l'espace de noms, on peut délimiter de manière définitive les fonctions de la gestion de noms. Ces fonctions sont les suivantes :

- Introduction, dans la hiérarchie de noms, d'un nom symbolique défini par un usager. Ceci est fait en introduisant le nom dans un répertoire. On apelle **cataloguer** cette opération. Une remarque importante : Ce sont les noms d'objet qui sont catalogués et non pas les objets. Le groupement d'objets est un aspect de la gestion d'objets complètement différent (cf. Conclusions).
- Création d'un répertoire. L'usager qui veut créer un répertoire doit lui attribuer un nom symbolique. Ce nom est catalogué dans un autre répertoire.
- Suppression d'un nom symbolique de la hiérarchie de noms. Ceci consiste à retirer le nom de son répertoire. Le nom d'un répertoire ne peut être suprimé que si le répertoire est vide. On parle alors de 'destruction d'un répertoire'.

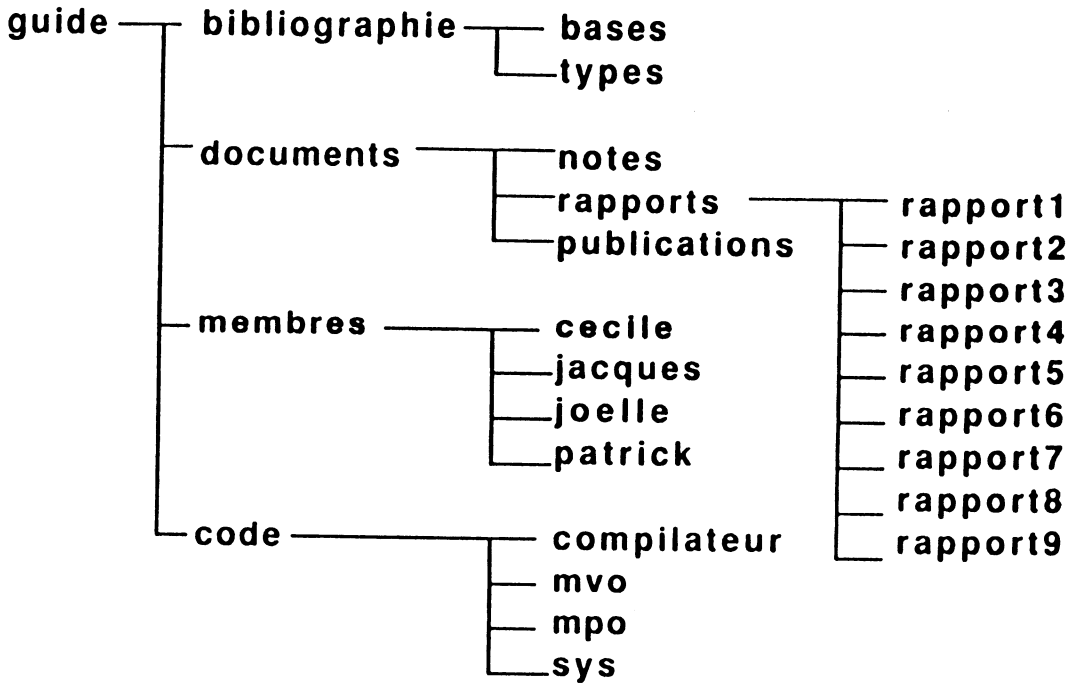


Figure 5.1 - La hiérarchie de noms symboliques

Rien n'empêche qu'un nom symbolique puisse être catalogué dans plusieurs répertoires.

A la racine de la hiérarchie se trouve le seul répertoire prédéfini du service de désignation symbolique. Il est aussi le seul qui existe dans l'espace de noms au moment de l'installation de ce service. Son nom symbolique est également prédéfini et il est connu par toutes les activités.

Le **chemin d'accès** d'un nom symbolique *n* relatif à un repertoire *x* est défini par une suite ordonnée de noms ayant les caractéristiques suivantes :

1. Le premier nom de la suite est catalogué dans le répertoire *x*.
2. Le dernier nom de la suite est *n*.
3. Le *i*ème nom de la suite est catalogué dans le répertoire dont le nom est le *(i - 1)*ème nom de la suite.

Un chemin d'accès est absolu lorsqu'il est relatif à la racine de la hiérarchie.

Deux exemples de chemin d'accès, définis à partir de l'espace de noms de la figure 5.1, sont les suivants : La suite {"rapports", "rapport7"} est le chemin d'accès au nom "rapport7" rélatif au répertoire de nom "documents". Par contre, la suite {"membres", "joelle"} constitue le chemin d'accès absolu au nom symbolique "joelle".

Le **parcours** (ou vérification) d'un chemin d'accès consiste à déterminer, pour tous le noms de la suite, si le *i*ème est le nom d'un répertoire qui catalogue effectivement le *(i + 1)*ème nom. Pouvant être catalogué dans plusieurs répertoires, tout nom

symbolique peut avoir plusieurs chemins d'accès et, par conséquent, plusieurs possibilités de parcours.

5.1.2 Les objets répertoires

Un répertoire est mis en oeuvre par un 'objet répertoire'(OR). Son type et classe, nommés `Directory` et définis en langage Guide, seront présentés dans une section ultérieure (cf. section 7.1). Le nom symbolique d'un OR est celui du répertoire correspondant.

Un OR comporte un ensemble d'associations <nom symbolique, référence-système>. L'association <*n*, *r*> appartient à l'état d'un OR si les deux conditions suivantes s'accomplissent :

1. Le nom symbolique *n* est catalogué dans le répertoire correspondant à l'OR.
2. *r* est une référence-système de l'objet dont le nom symbolique est *n*. Il peut s'agir, bien évidemment, de la référence-système d'un OR.

La figure 5.2 illustre les ORs correspondants à quelques-uns des répertoires de la figure 5.1.

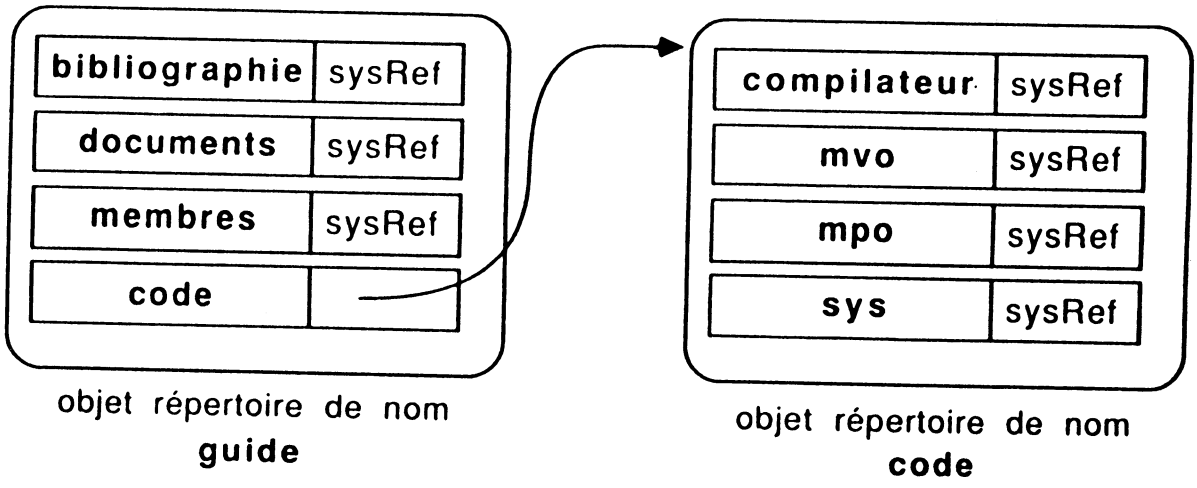


Figure 5.2 - Objets répertoires

L'état des ORs comporte les informations nécessaires pour la réalisation de la gestion de noms et de la désignation symbolique d'objets, les deux groupes de fonctions du service de désignation symbolique. Ce service est ainsi mis en oeuvre par les méthodes des ORs.

Une fonction de la gestion de noms est la création de répertoires. Or, la création d'un répertoire entraîne la création de l'OR correspondant. Puisque les ORs mettent en oeuvre la gestion de noms, c'est par le biais de leurs méthodes que sont créés les autres

ORs. Seulement l'OR correspondant à la racine n'est pas créé de cette manière.

En effet, au moment de son installation, le service de désignation symbolique est constitué d'un seul et unique OR vide qui correspond à la racine. La référence-système de cet objet est connue de toutes les activités.

Soit x un OR. Ses méthodes réalisent les fonctions suivantes :

- Donner un (autre) nom symbolique à un objet dont on connaît la référence-système. Ceci correspond à une fonction de la gestion de noms et à une fonction de la désignation symbolique d'objets :
  - — Le nom est catalogué dans le répertoire correspondant à x.
  - — L'association du nom et de la référence-système de l'objet est conservée dans l'état de x.
- Obtenir la référence-système d'un objet dont on connaît le nom symbolique.
- Détruire le nom symbolique d'un objet. S'il s'agit du nom d'un répertoire non vide, la fonction échoue. La destruction du nom d'un objet correspond à une fonction de la gestion de noms et à une fonction de la désignation symbolique d'objets :
  - — Le nom est supprimé du répertoire correspondant à x.
  - — L'association du nom et de la référence-système de l'objet est retirée de l'état de x.

  Le fait que le nom supprimé soit le dernier du répertoire correspondant à x n'implique pas que ce répertoire soit détruit. Pour faire ceci, il est nécessaire de supprimer le nom de x. Cette fonction doit être réalisée par l'OR correspondant au répertoire où se trouve catalogué le nom de x.
- Changer le nom symbolique d'un objet par un autre.
- Créer un nouveau répertoire. Un nom symbolique doit lui être attribué. Un nouvel OR est créé et, pour des raisons qu'on éclaircira plus tard (cf. ci-après), une référence-système de l'objet x est copiée dans son état. Après la création de ce nouvel objet et l'obtention de sa référence-système, tout se passe comme si on donnait un nom symbolique à un objet dont on connaît la référence-système (première fonction).
- Créer un 'lien' vers un répertoire. Fonction héritée de la gestion d'espaces hiérarchiques de noms de fichiers, elle consiste à créer et nommer un répertoire doté d'un mécanisme qui le lie à un autre répertoire, disons r. Cette liaison permet de considérer que tous les noms symboliques catalogués dans r sont aussi catalogués dans le répertoire créé. Du point de vue de la gestion de noms, la fonction consiste simplement à donner un nouveau nom à r.

Afin de rendre plus souple le service de désignation symbolique, il a été décidé qu'un OR (disons x) puisse réaliser (ou faire réaliser) les fonctions antérieures sur quatre groupes de noms symboliques :

1. Les noms catalogués dans le répertoire correspondant à *x*.
2. Les noms catalogués dans le répertoire où le nom de *x* est lui-même catalogué.
3. Les noms dont le chemin d'accès est absolu.
4. Les noms dont le chemin d'accès est rélatif au répertoire correspondant à *x*.

Dans le premier cas, l'objet *x* réalise les fonctions directement.

Dans le deuxième cas, les fonctions doivent être réalisés par l'OR correspondant au répertoire qui catalogue le nom de *x*. La référence-système de cet objet a été placée dans l'état de *x* au momet de sa création (cf. ci-dessus). Il suffit à *x* d'appeler les méthodes de cet OR.

Dans le troisième cas, c'est l'OR correspondant à la racine qui doit réaliser effectivement les fonctions. Sa référence-système de celui-ci est connue par l'activité qui exécute les méthodes de *x*.

Dans le quatrième cas, un parcours de chemin est nécessaire pour déterminer l'OR correspondant au répertoire qui catalogue le dernier nom du chemin. C'est lui qui doit réaliser effectivement les fonctions.

Le parcours du chemin d'accès est mis en oeuvre de la manière suivante : *x* réalise la fonction d'obtention de la référence-système associée à un nom (deuxième fonction de la liste ci-dessus) afin d'obtenir la référence de l'OR dont le nom est le premier du chemin. Cet objet réalise la même fonction afin d'obtenir la référence de l'OR dont le nom est le deuxième du chemin. Ceci continue jusqu'à l'obtention de la référence de l'OR correspondant au répertoire où se trouve le dernier nom du chemin.

Toute fonction du service de désignation symbolique comporte donc deux parties :

1. Détermination de l'OR qui doit réaliser effectivement la fonction.
2. Réalisation de la fonction par cet OR.

5.1.3 La désignation symbolique

Un objet peut être désigné de quatre manières :

1. Par son nom.
2. Par le chemin d'accès de son nom relatif à un répertoire quelconque.
3. Par le chemin d'accès absolu de son nom.
4. Par un mécanisme qui indique qu'il est un OR dont le répertoire correspondant catalogue le nom d'un autre objet.

Des outils syntaxiques ont été introduits pour permettre aux usagers d'exprimer ces désignations. Ces outils sont :

- Le caractère '/'. Il est utilisé comme séparateur entre les noms symboliques que comporte un chemin d'accès. Il est aussi le nom prédéfini de la racine de la hiérarchie.
- Le caractère '.'. Par rapport à un répertoire de la hiérarchie, il désigne l'OR correspondant.
- La chaîne de caractères '..'. Par rapport à un répertoire *r* de la hiérarchie, elle désigne l'OR qui a créé l'OR correspondant à *r*.

Quelques exemples illustrent l'utilisation de ces outils syntaxiques. L'espace de noms est celui de la figure 5.1.

- La chaîne "/membres/joelle" représente le chemin d'accès absolu du nom symbolique "joelle".
- La chaîne "rapports/rapport7" correspond au chemin d'accès du nom "rapport7" relatif au répertoire de nom "documents".
- La chaîne "../documents" sert à désigner l'OR de nom "documents" à partir de l'OR de nom "rapports".
- La chaîne "./mvo" désigne l'objet de nom "mvo" à partir de l'OR de nom "code". On remarque que la chaîne "mvo", qui correspond au chemin d'accès du nom "mvo" relatif au répertoire de nom "code", produit le même effet.

Les quatre désignations de base peuvent se combiner pour produire des désignations symboliques plus complexes. Un exemple illustre la combinaison de désignations : Toujours par rapport à la hiérarchie de la figure 5.1, la chaîne "../../membres/joelle" sert à désigner l'objet de nom "joelle" à partir de l'OR de nom "rapports".

Toute forme de désignation, de base ou complexe, s'exprime par le moyen d'une 'expression de désignation'. Il s'agit de chaînes de caractères qui résultent de la combinaison de noms symboliques et des outils syntaxiques antérieurs afin de désigner un objet. Les méthodes des ORs ne désignent les objets que par le biais des expressions de désignation.

La définition formelle des expressions de désignation est la suivante :

```
expression-de-désignation ::=
  / |
  /expression-1 |
  expression-1

expression-1 ::=
 nom-de-noeud |
 nom-de-noeud/expression-1
```

```
nom-de-noeud ::=
 nom-symbolique |
 .. |
 .
```

Les noms symboliques sont des chaînes de caractères définies par l'usager. En principe, ils sont de taille quelconque.

Il est prévu en outre que l'interface d'utilisation de Guide (cf. section 3.5) permette d'appeler le service de désignation symbolique de manière interactive. Dans ce cas, le service doit offrir la possibilité d'introduire des 'atouts' dans ses expressions de désignation.

Un atout (*wildcard*) est une chaîne de caractères utilisée comme filtre pour sélectionner un sous-ensemble des noms catalogués dans un répertoire. Par exemple, l'expression de désignation "/documents/rapports/rapport[1-3]" permet de désigner les trois premiers rapports de la hiérarchie de la figure 5.1.

Dans la définition formelle d'une expression de désignation, un atout constitue une autre option des `nom-de-noeud`. Les conventions du 'Bourne *shell*' [Bourne78] sont utilisées pour la définition d'atouts.

5.2 Le service de versions

Un modèle de données à objets fait des objets les unités de manipulation des applications. Du point de vue de celles-ci, pourtant, les objets ne sont pas fixes : ils peuvent être visualisés de façons différentes. Les 'versions d'un objet' sont les manières différentes de présenter cet objet.

Par exemple, les éditions française et anglaise sont deux façons de présenter le même livre. En conséquence, elles sont des versions d'un objet de type 'livre'. Egalement, une chanson peut être présentée de trois façons différentes : Les paroles de la chanson, qui constituent un objet de type 'chaîne de caractères', sa partition, qui est un objet de type 'graphique', et sa mélodie, qui donne lieu à un objet de type 'son'.

Un ensemble de versions particulier est celui qui reflète l'évolution dans le temps d'un objet. Pour les applications qui prennent en compte les étapes de la vie d'un objet, les phases de l'évolution de l'objet constituent ses versions.

En Guide, les caractéristiques des versions des objets ont été déterminées en fonction des besoins des applications. Le génie logiciel et la gestion de documents, applications pilotes du système [Guide-R1] [Quint86], envisagent les considérations suivantes en ce qui concerne le traitement des versions :

- Un objet évolue dans la mesure où son état subit des modifications. A la fin de certaines étapes de cette évolution, l'état de l'objet doit être sauvegardé en vue d'une utilisation future. Une version doit donc être créée pour conserver cette sauvergarde.

- Les modifications sur une version ne sont pas interdites. Si elle est modifiée, une version évolue et, à un moment donné, il peut être nécessaire de sauvegarder son contenu. Ceci revient à créer une version à partir d'une autre.
- Dans certains cas, l'accès aux versions peut se réaliser selon l'ordre imposé par les dates de création de celles-ci.

Ces considérations nous ont permis de caractériser les versions des objets de la manière suivante :

1. Tout objet persistant peut posséder des versions.
2. Une nouvelle version d'un objet est créée à la demande explicite des applications. La production de versions n'est pas automatique.
3. La création d'une nouvelle version d'un objet se réalise en sauvegardant le contenu d'une version ancienne. Cette sauvegarde doit avoir lieu de manière telle qu'on puisse accéder au contenu de la nouvelle version et à celui de l'ancienne.
4. Une version d'un objet peut être modifiée à tout moment.
5. L'historique des versions d'un objet doit être maintenu afin que les applications puissent accéder à celles-ci selon l'ordre qu'imposent leurs dates de création.
6. Une version d'un objet est détruite à la demande explicite des applications.

Un autre service du système Guide, le service de gestion de versions des objets, prend en charge la gestion de l'ensemble des versions de chaque objet. Les fonctions de base de ce service sont les suivantes :

- Créer une nouvelle version d'un objet.
- Localiser une version particulière d'un objet.
- Détruire une version d'un objet.

Une fois localisée, une version peut être consultée et/ou modifiée par les applications.

5.2.1 Création de versions

Chaque version d'un objet est elle-même un objet. Ce choix est justifié par les raisons suivantes :

- Eviter de gérer des listes de versions à l'intérieur de la machine virtuelle.
- Eviter d'allonger l'oid d'un objet en rajoutant un identificateur de version.

On a établi que la création d'une nouvelle version se fait par sauvegarde du contenu d'une ancienne. Puisque les versions sont des objets, ce qu'on sauvegarde d'une version pour en créer une autre n'est autre chose que l'état de l'objet qui la met en oeuvre. En conséquence, les objets qui mettent en oeuvre les versions d'un objet sont tous des instances de la même classe.

La toute première version d'un objet est obtenue par sauvegarde de l'état de cet objet. La deuxième version peut ensuite être créée soit à partir de l'objet, soit à partir de la première version. Postérieurement, les nouvelles versions sont créées soit à partir de l'objet, soit à partir des versions déjà existantes.

Trois remarques découlent des affirmations antérieures :

1. Avant toute création de versions d'un objet, l'objet est lui-même sa seule et unique version.
2. Par rapport à une nouvelle version d'un objet, l'objet lui-même est une version ancienne.
3. Les objets qui mettent en oeuvre les versions d'un objet et cet objet lui-même sont tous des instances de la même classe.

Par un souci d'uniformité avec les manières de désignation de versions qu'on verra par la suite (cf. section 5.2.2), on dit qu'un objet est sa propre version 0. Le terme 'versions d'un objet' comprend désormais l'objet lui-même et toutes ses versions.

La création d'une nouvelle version à partir d'une ancienne se fait en deux étapes :

- Une nouvelle instance de la classe de l'ancienne version est créée.
- Dans l'état de cette nouvelle instance est copié l'état de la version ancienne.

5.2.2 Désignation de versions

Quelques caractéristiques des versions permettent d'introduire des ordres à l'intérieur de l'ensemble des versions d'un objet. Ces ordres nous permettront de désigner univoquement chaque version. Ces caractéristiques sont les suivantes :

- La manière dont les versions sont créées et modifiées.
- L'accès aux versions selon l'ordre de l'historique.

Le fait qu'on crée une version à partir d'une autre (et seulement à partir d'une autre) établit une structure arborescente entre les versions. Cette structure, qu'on appelle **arbre de dérivation**, permet d'introduire une première façon de désignation de versions. Ceci a été entrepris de la manière suivante :

Considérons deux niveaux de l'arbre de dérivation d'un ensemble de versions.

Dans le graphe de la figure 5.3, qui représente ces deux niveaux, les conventions suivantes ont été suivies :

1. Chaque noeud correspond à une version différente.
2. Chaque version possède son propre oid[13].

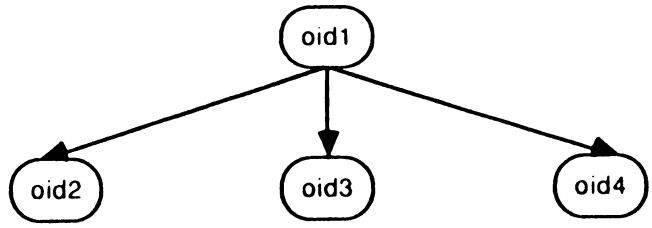


Figure 5.3 - Arbre de dérivation

3. L'orientation des arcs met en évidence le mécanisme de création de versions : Les versions du niveau inférieur ont été créées à partir de celle du niveau supérieur.

Si, dans le niveau inférieur, les versions sont ordonnées selon leurs dates de création, elles peuvent être désignées par le moyen d'un identificateur dont le contenu suit cet ordre. Ainsi, l'objet dont le oid est `oid2` est la première (i.e., la plus ancienne) des versions créées à partir de l'objet dont l'oid est `oid1` et l'objet dont l'oid est `oid4` est la dernière (i.e., la plus récente) de ces versions.

Le fait qu'on puisse créer de versions à partir d'une version antérieure est pris en compte en ajoutant un nouveau niveau à l'arbre. On appelle **mère** la version à partir de laquelle sont créées les versions du nouveau niveau. Dans ce niveau, l'ordre de création est également pris en compte pour désigner les versions.

Si, à l'intérieur de chaque niveau de l'arbre de dérivation, l'on ajoute l'identificateur de la mère *à la fin* de celui de chacune des versions, il est possible de désigner univoquement toutes les versions d'un objet. La désignation de versions par le biais des arbres de dérivation est illustrée par la figure 5.4. On peut remarquer qu'il ne s'agit pas de la désignation 'classique' des noeuds d'un arbre.

Contrairement à l'arbre de dérivation, la linéarité de l'historique des versions d'un objet permet un schéma de désignation simple, c'est-à-dire, dépourvu de niveaux. La première version dans l'historique est la plus ancienne et la dernière version dans l'historique est la plus récente. De même, l'historique permet de désigner les versions en séquence : A partir d'une version quelconque, il est possible de désigner la suivante ou la précédente.

La désignation de versions par le biais d'un historique est illustrée par la figure 5.5.

Les versions d'un objet peuvent donc être désignées de deux manières :

---

13. Par simplicité, on utilise, dans cette section, les oids des objets plutôt que ses références-système.

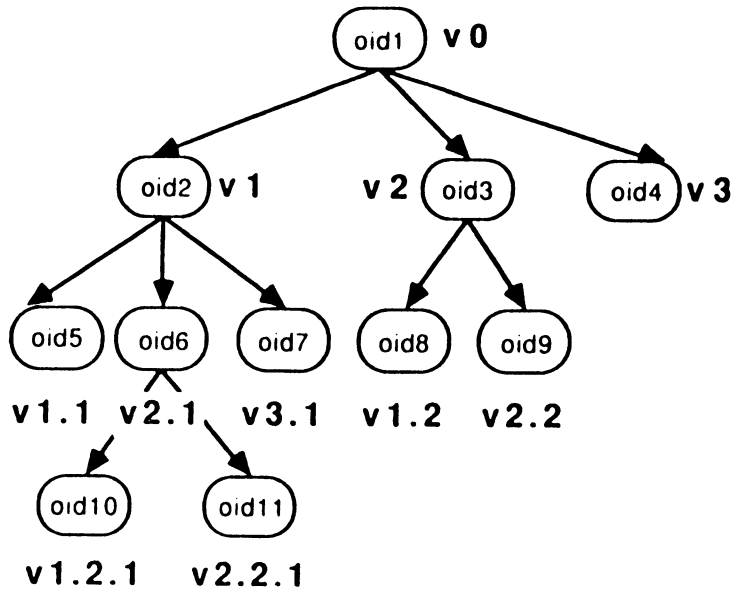


Figure 5.4 - Désignation de versions en utilisant l'arbre de dérivation

1. En utilisant l'arbre de dérivation, une version d'un objet est désignée par l'expression `N[.exp]`, où `N` indique l'ordre de création de la version par rapport aux autres filles de sa mère et `exp` est l'expression qui désigne la mère. Lorsque `N = 0` et `exp = NIL`, la version désignée est l'objet lui-même.
2. En utilisant l'historique, une version est désignée par les expressions suivantes :
   - `first` : Il s'agit de la version la plus ancienne.
   - `last` : Il s'agit de la version la plus récente.
   - `next(exp)` : La version qui, dans l'historique, suit (a été créée après) la version désignée par l'expression `exp`. Aucune version n'est désignée si `exp = last`.
   - `previous(exp)` : La version qui, dans l'historique, précède (a été créée avant) la version désignée par l'expression `exp`. Aucune version n'est désignée si `exp = first`.

Quelques exemples illustrent l'utilisation de ces expresions. Les versions désignées sont celles des figures 5.4 et 5.5.

- L'expression `2.2.1.0` désigne l'objet dont l'oid est `oid11`.
- L'expression `previous(next(previous(previous(last))))` désigne l'objet dont l'oid est `oid10`.
- Si `exp` est une expression qui désigne l'objet dont l'oid est `oid2`, l'expression `2.2.exp` désigne alors l'objet dont l'oid est `oid11`.

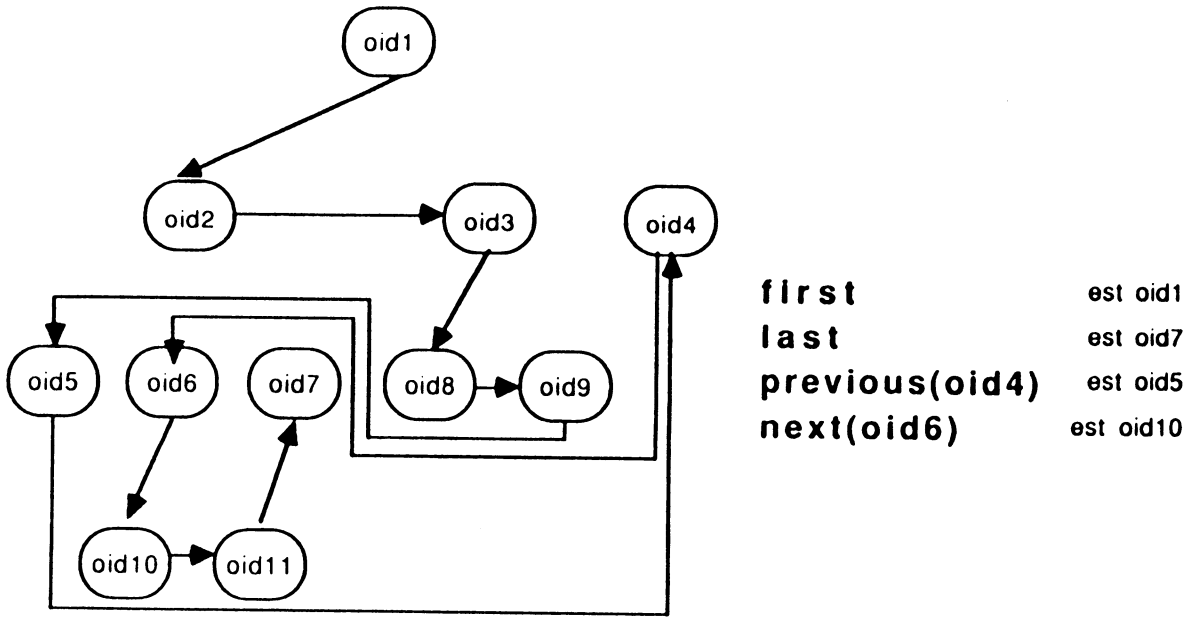


Figure 5.5 - Désignation de versions en utilisant l'historique

- Si `exp` est une expression qui désigne l'objet dont l'oid est `oid1`, l'expression `2.2.exp` désigne alors l'objet dont l'oid est `oid9`.
- Si `exp` est une expression qui désigne l'objet dont l'oid est `oid9`, l'expression `previous(next(previous(previous(exp))))` désigne alors l'objet dont l'oid est `oid3`.

Les trois derniers exemples montrent que les deux désignations de base peuvent se combiner pour produire des désignations plus complexes. Un exemple illustre la combinaison de désignations : Les versions désignées étant toujours celles des figures 5.4 et 5.5, l'expression

```
previous(next(previous(previous(2.2.first))))
```

désigne l'objet dont l'oid est `oid3`.

Toute forme de désignation, de base ou complexe, s'exprime par le moyen d'une 'expression de désignation'. Il s'agit de chaînes de caractères qui résultent de la combinaison des expressions antérieures afin de désigner une version. Les versions sont désignées toujours par le biais des expressions de désignation.

La définition formelle des expressions de désignation est la suivante :

```
expression-de-désignation ::=
 first |
 last |
 next(expression-de-désignation) |
 previous(expression-de-désignation) |
 expression-arbre-derivation |
 expression-arbre-derivation.expression-de-désignation

expression-arbre-dérivation ::=
 cardinal |
 cardinal.expression-arbre-dérivation
```

Le terme `cardinal` est utilisé ici pour indiquer toute chaîne de caractères construite exclusivement à partir des caractères correspondants aux neuf chiffres et au zéro. Ces chaînes sont définies par les usagers du service de versions.

5.2.3 Le gestionnaire de versions

La gestion des versions d'un objet est réalisée par le moyen d'un nouvel objet, appelé 'gestionnaire de versions' (GV). Son type et sa classe, nommés `VersManager` et définis en langage Guide, seront présentés dans une section ultérieure (cf. section 7.2).

Tout GV comporte une fonction interne qui, à partir d'une expression de désignation du service de versions, permet à tout instant de savoir si celle-ci est valide ou non, et, si oui, quelle est la référence-système de la version désignée. Une fois la référence en main, toutes les manipulations sur la version sont possibles.

Les méthodes d'un GV réalisent les fonctions suivantes :

- Localiser la version désignée par une expression de désignation. L'application de sa fonction interne permet au GV d'obtenir la référence-système de la version.
- Créer une nouvelle version à partir de la version désignée par une expression de désignation. Le GV vérifie que l'ancienne version existe. Il utilise pour ceci sa fonction interne. Il crée ensuite la nouvelle version.
- Détruire la version désignée par une expression de désignation. L'application de sa fonction interne permet au GV de déterminer que la version qu'on détruit existe. Il détruit ensuite la version désignée et toute les versions créées à partir de celle-ci.

La destruction d'une version peut être à l'origine d'une réorganisation de l'ensemble des versions. Si, par exemple, la version désignée par l'expression `1.1.0` est détruite, les versions désignées par `2.1.0` et par `3.1.0` doivent désormais être désignées, respectivement, par les expressions `1.1.0` et `2.1.0`.

Le service de versions est ainsi mis en oeuvre par les méthodes des objets GVs.

La liaison entre un objet et un GV est mise en oeuvre en enrichissant la classe `Top` de deux particularités de gestion de versions. Les objets étant tous des instances de cette classe [Guide-R2], la gestion de versions est garantie à chacun. Ces deux particularités sont les suivantes :

1. L'état de chaque objet est augmenté d'une référence-système vers le GV qui gère ses versions. Si l'objet n'a pas de versions, cette référence est nulle. Les versions étant elles-mêmes des objets, elles comportent aussi une référence vers un GV. Cependant, afin de maintenir un seul GV par ensemble de versions, toute version d'un objet *x* contient une référence vers le GV associé à *x*. La figure 5.6 montre l'ensemble des versions d'un objet et le GV associé.

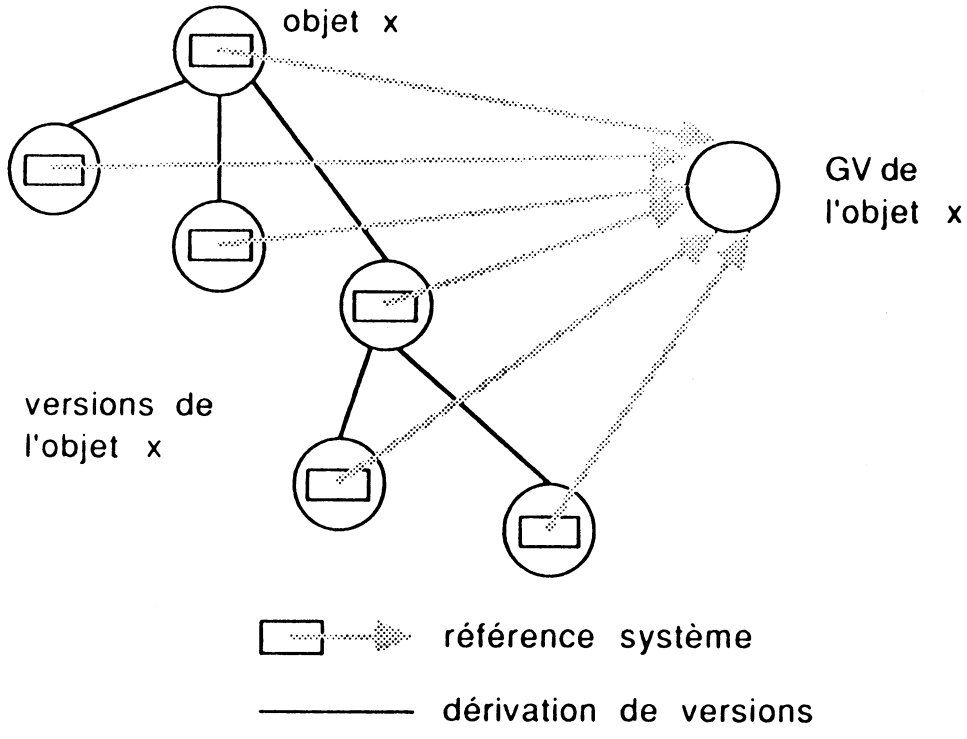


Figure 5.6 - Le GV d'un objet

2. On ajoute trois méthodes à chaque objet : Création, destruction et obtention des versions. En fait, ces méthodes font des appels aux méthodes du GV associé pour la réalisation des fonctions correspondantes. Ce mécanisme présente deux avantages :

   - Les applications peuvent traiter un objet, en même temps, comme instance de sa classe et comme entité ayant (ou pouvant avoir) des versions.
   - Les applications peuvent manipuler les versions de chaque objet de manière 'naturelle', c'est-à-dire, sans avoir à connaître l'existence du GV.

La première action de la méthode de création de versions est l'analyse du contenu de la référence vers le GV. Seulement si la elle est égale à NIL, une nouvelle instance de la classe `VersManager` est créée. On remarque que ceci n'arrive que lors de la création de la toute première version d'un objet. En effet, la référence n'aura plus jamais la valeur NIL. Etant donné que les nouvelles versions sont créées par copie de l'état des anciennes, le contenu de la référence vers le GV est copié dans toutes les versions.

Toute version doit pouvoir se désigner elle même. L'expression de désignation `self` est utilisée pour celà. Cette expression peut se combiner avec les autres, comme l'illustrent les exemples suivants :

```
2.1.self et 3.1.previous(self).
```

5.3 Caractéristiques communes aux services

D'un point de vue fonctionnel, les services de désignation symbolique et de gestion de versions présentent un ensemble de caractéristiques communes :

- Ils sont tous les deux réalisés par le moyen des méthodes d'un ensemble d'objets et non pas par un domaine serveur.
- Ils mettent en oeuvre tous les deux un groupement 'logique' d'objets. Le service de versions regroupe les versions d'un objet. Le service de désignation symbolique regroupe les objets possédant un nom symbolique. Selon le niveau où l'on se place dans la hiérachie de l'espace de noms, ce deuxième groupement comporte un nombre plus ou moins grand d'objets.
- A l'intérieur de ces groupes, les objets sont désignés par le biais d'expressions de désignation plus ou moins complexes. Les services se comportent donc comme des fonctions qui, à partir d'une expression de désignation, retournent la référence-système de l'objet désigné.
- Les fonctions que les services réalisent sont similaires :
  - — Définition (création) de l'entité qui regroupe.
  - — Association d'une référence-système à une expression de désignation (prise en compte d'un nom symbolique ou création d'une version).
  - — Récupération de la référence-système d'un objet en utilisant une expression de désignation.
  - — Destruction de l'association existante entre une référence-système et une expression de désignation.

Une dernière caractéristique commune aux deux services est la manière dont ils réalisent le contrôle d'accès. Une expression de désignation permet d'obtenir une référence-système d'un objet et, par conséquent, d'en invoquer les méthodes. Dans la mesure où les objets sont protégés par des mécanismes de contrôle des droits d'accès

[Guide-R4], il ne semble pas utile à première vue d'associer un mécanisme de protection aux services.

Dans certains cas, il peut être nécessaire d'accéder à un objet par le moyen d'un autre. Prenons l'exemple d'un livre composé de chapitres : On peut donner le droit à un utilisateur d'invoquer certaines méthodes sur les chapitres via les méthodes sur le livre, sans pour autant lui donner le droit d'appeler directement les mêmes méthodes sur les chapitres. S'il n'est pas possible de désigner par une expression les objets composants, aucun problème ne se pose : l'utilisateur ne peut pas obtenir directement une référence sur un chapitre. Par contre, si une expression de désignation permet de désigner un objet composant, rien n'empêche un utilisateur quelconque d'obtenir une référence sur le chapitre, et d'invoquer directement une méthode.

Le fait d'associer un contrôle d'accès à chaque service permet de résoudre ce problème. L'accès aux ORs et aux GVs doit donc être protégé. Les mécanismes de protection standard du système sont utilisés. Ces mécanismes sont basés sur des objets 'listes d'accès' qui restreignent les droits de certains utilisateurs, ces droits étant définis sous forme de 'méthode autorisée ou interdite'. La protection dans le système Guide est décrite en détail dans [Guide-R4].

6. Réalisation du modèle de la mémoire

La machine virtuelle de Guide réalise les fonctions suivantes :

1. La gestion de domaines et d'activités.
2. La gestion d'images.
3. La gestion d'objets stockés.

La gestion de domaines et d'activités relève du modèle d'exécution. Elle ne sera présentée dans ce chapitre que de manière superficielle, afin de montrer son rapport avec les autres gestions au sein de la machine virtuelle. Les détails de l'implantation du modèle d'exécution se trouvent dans [Guide-R3] et [Guide-R5]. La gestion d'images et la gestion d'objets stockés constituent l'implantation du modèle de la mémoire. Leur description est donc l'objectif de ce chapitre.

La réalisation des modèles d'exécution et de la mémoire met en évidence la nature répartie du système Guide. Bien que, pour les applications programmées en langage Guide, l'accès aux objets distants et locaux est transparent, la mise en oeuvre de la machine virtuelle doit prendre en compte la notion de site. Un **site** Guide est une machine support du système Guide.

6.1 Description d'un site Guide

La machine virtuelle est mise en oeuvre sur l'ensemble des sites Guide. La mémoire fictive d'un site[14], augmentée des fonctions de gestion d'images, participe à la réalisation d'une ou plusieurs mémoires virtuelles de la machine à objets. De même, une famille de processus, s'exécutant sur des sites différents, est utilisée pour la réalisation répartie d'un processeur virtuel (cf. section 6.2).

La mémoire secondaire d'un site est dédiée à la réalisation d'un ou plusieurs systèmes d'objets. Ceci est fait en l'enrichissant avec les fonctions de gestion d'objets stockés.

Un système d'objets peut s'étaler sur plusieurs supports de mémoire secondaire, ce qui n'est pas le cas des volumes qui le composent. En effet, la taille des supports de mémoire secondaire étant de plus en plus importante, la possibilité d'implanter un volume sur plus d'un support n'a pas été prévue.

---

14. On appelle 'mémoire fictive' d'une machine soit sa mémoire physique, soit la mémoire virtuelle fournie par un système support.

La figure 6.1 illustre la mise en oeuvre de la machine à objets sur l'ensemble des sites Guide. On constate que les systèmes d'objets peuvent être multi-supports et multi-sites mais les volumes non. Sur les sites dont la mémoire secondaire n'est pas dédiée au rangement d'objets stockés, il sera possible, comme l'on verra plus loin (cf. section 6.3.2), de créer des images à partir des blocs des objets stockés distants.

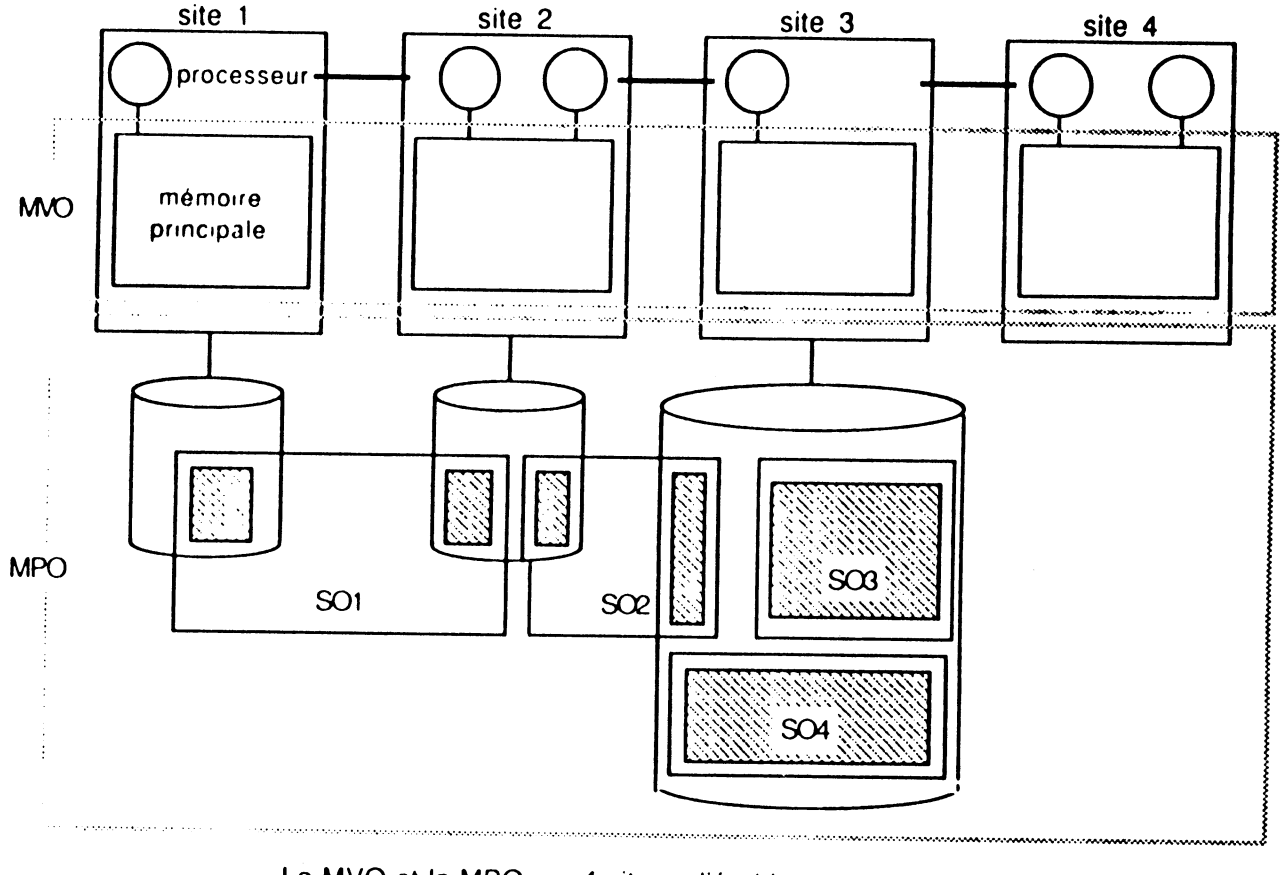


Figure 6.1 - Implantation de la machine à objets sur les sites Guide

La réalisation répartie de la gestion de domaines et activités, couche supérieure de la machine virtuelle, est entreprise en utilisant les processus des sites ainsi que leurs mécanismes de communication et synchronisation. La figure 6.2 montre une première approche de la mise en oeuvre des domaines et des activités sur l'ensemble des sites Guide.

Les images, les objets stockés et les volumes sont des entités mono-sites. Les domaines, les activités et les systèmes d'objets sont des entités multi-sites. Ce choix a été motivé par la considération suivante : la gestion d'une entité multi-sites pose de problèmes délicats, tels que la cohérence et la reprise ; on a préféré traiter ces problèmes au niveau de mécanismes relativement 'gros grain', comme les domaines, les activités et les système d'objets. Pour les entités 'petites', telles que les images, les

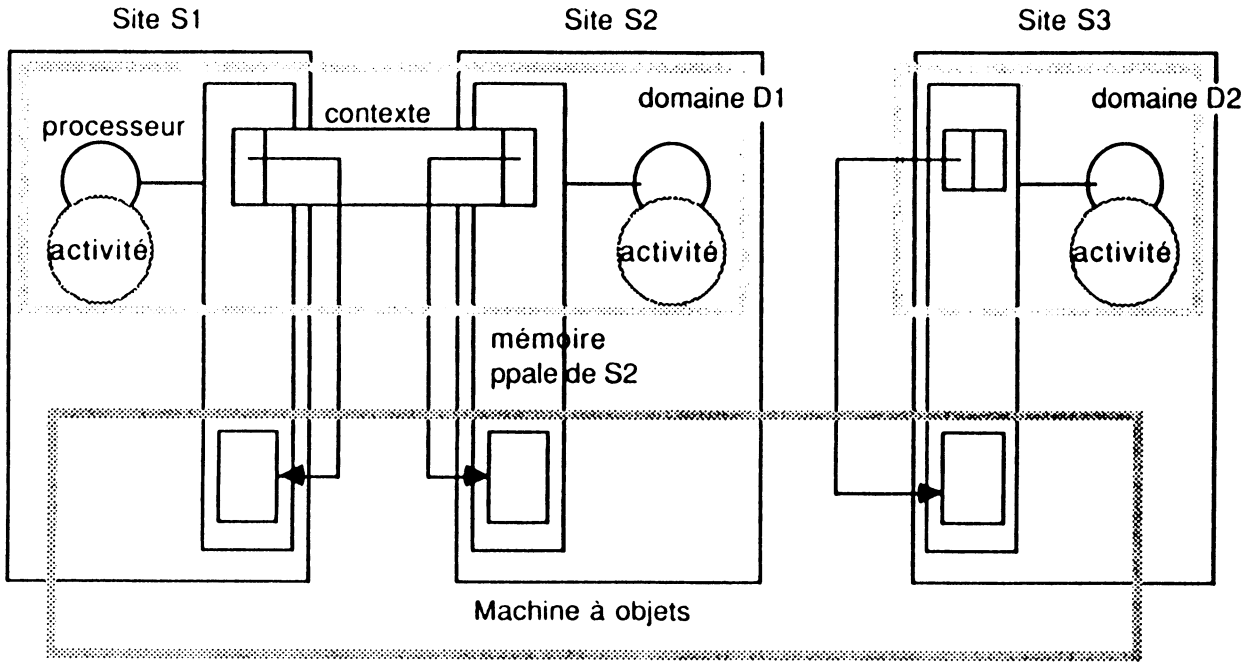


Figure 6.2 - Implantation des domaine et des activités sur les sites Guide

objets stockés et les volumes, la gestion a donc été simplifiée.

Le verrouillage d'objets a été défini en termes d'une mémoire virtuelle de la machine à objets (cf. section 4.1.1.1). Pourtant, puisque les images sont des entités mono-sites, il est nécessaire de restreindre le verrouillage à la mémoire fictive d'un site. Ainsi, lorsqu'un objet est verrouillé sur la mémoire d'un site (ou, simplement, sur un site), une image de l'objet ne peut être créée sur aucun autre site.

Chaque site est identifié de manière unique. Les sites sont reliés entre eux par un réseau de communication sur lequel sont disponibles des primitives de communication du niveau transport [Laforgue88].

6.1.1 Le système support

Pour la mise en oeuvre des fonctions du système sur une machine physique, les concepteurs et réalisateurs de Guide avaient deux voies à suivre :

1. Utilisation d'un outil existant qui puisse servir de support à l'implantation des modèles d'exécution et de la mémoire. En ce qui concerne le premier modèle, cet outil se chargerait de la gestion des processeurs et processus du site. Pour le compte du deuxième, il réaliserait la gestion de l'espace réservé pour les objets. Il fournirait en outre les moyens de communication fiables entre sites.

2. Construction sur machine nue de cet outil.

Etant donné que la conception et l'implantation d'un tel outil ne fait pas partie des objectifs immédiats de Guide et aussi pour des raisons d'économie de temps, nous avons décidé de prendre la première voie. Nous avons choisi le système Unix V.2 comme outil de support à notre système. La première version du système Guide a été mise en oeuvre sur deux des versions du système Unix V.2 : Spix 30 sur SPS-7/300 et SunOS 3.5 sur Sun-3.

L'utilisation des primitives de la mémoire partagée d'Unix[15] pour la mise en oeuvre de la MVO sur un site permet de réaliser l'un des objectifs capitaux de Guide : le partage des objets. Cependant, quelques fonctions d'Unix présentent des inconvénients. Par exemple, la désignation de fichiers ne s'avère pas très efficace pour un système où l'objet est en même temps l'unité de stockage et l'unité d'exécution. De même, le système d'entrées/sorties n'a pas été conçu pour supporter des applications requérant un grand niveau de fiabilité. Pour ces raisons, on a préféré réaliser, comme le font certains systèmes analysés dans [Exertier87], l'accès à la mémoire secondaire en mode *raw*. On ne fait donc pas correspondre un fichier Unix à un objet stocké Guide.

Certaines fonctions de support à la gestion d'objets ont été mises en oeuvre par des modules construits au-dessus d'Unix. Parmi ces fonctions, on peut citer la gestion du code exécutable, le transfert de l'état d'un objet, la manipulation de l'espace en mémoire, la mise en oeuvre de la synchronisation, la détection des modifications à l'état d'un objet, etc. N'ayant pas l'intention de décrire ici tous ces modules, on se restreint à ceux qui servent de support à la gestion de la mémoire d'objets. Les détails des autres modules se trouvent dans [Guide-R5], [Guide-R7], [Decouchant88a] et [Meysembourg88].

Le support à la gestion d'images consiste à organiser la mémoire partagée disponible de manière à faciliter la création et la destruction d'images. Le support à la gestion d'objets stockés utilise les primitives de l'interface-disque en mode *raw* d'Unix pour organiser la mémoire secondaire et faciliter ainsi l'accès aux blocs d'un objet stocké.

6.1.2 Support à la réalisation d'images

Ce module réalise deux fonctions : L'implantation des segments des images créées sur le site et la gestion des adresses d'attachement de ces segments à l'intérieur des activités qui se déroulent sur le site.

Des segments de mémoire partagée d'Unix sont utilisés pour l'implantation des segments des images. La mise en oeuvre de la correspondance entre ces deux sortes de

15. On substitue désormais le terme 'Unix' aux termes 'Spix 30' et 'SunOS 3.5'.

segments doit prendre en compte deux restrictions imposées par la manière dont les segments de mémoire partagée sont réalisés par le système Unix sous-jacent :

1. La mémoire partagée, dans la version SunOS 3.5, n'est pas paginée. La zone allouée à la mémoire partagée est un paramètre de génération d'Unix : Plus cette zone est grande, plus la mémoire physique disponible pour la pagination est petite et donc plus les risques d'écroulement des performances du système sont élévés.
2. Pour un segment, l'allocation minimale de mémoire partagée sur SunOS 3.5 doit être un multiple de 8 KOctets. Cette taille est peu adaptée à la gestion de petites images.

A cause de ces restrictions, deux mises en oeuvre parallèles ont été effectuées pour ce module. Dans la première, chaque segment d'image est réalisé par un segment de mémoire partagée. Dans la seconde, un seul segment de mémoire partagée implante tous les segments des images.

La gestion des adresses d'attachement diffère aussi dans les deux mises en oeuvre. En effet, si l'on ne compte que sur un seul segment de mémoire partagée, toutes les activités seront obligées de l'attacher à une adresse virtuelle, et ceci même si elles n'accèdent qu'à un seul objet. Par contre, dans le cas de la première réalisation, les activités n'attachent que les segments de mémoire partagée qui implantent les segments d'image des objets auxquels elles accèdent.

6.1.2.1 Version mono-segment de la gestion de la mémoire partagée

Les segments des images créées sur un site se trouvent tous dans un segment de mémoire partagée Unix. On fait donc correspondre un **bloc** du segment de mémoire partagée à chacun des segments d'image.

La création et destruction de blocs de mémoire partagée sont mises en oeuvre en utilisant l'algorithme du *buddy* généralisé [Peterson77]. Cet algorithme partitionne la mémoire libre en ensembles de blocs de taille fixe. Il permet, en plus, de recomposer rapidement les blocs de mémoire libérés en blocs de taille plus grande.

Pour la création du segment de son image, un objet se voit allouer un bloc de mémoire partagée dont la taille est exactement celle de son état ou celle immédiatement supérieure, si la taille de l'état n'est pas dans la suite des tailles gérées par le *buddy*. L'allocation a lieu lors du verrouillage de l'objet sur le site. Le bloc reste alloué jusqu'à ce que, pour des raisons de manque d'espace, l'image soit détruite.

Toutes les activités du site attachent à la même adresse virtuelle le segment de mémoire partagée. Ceci fait que tous les segments d'image soient présents, à la même adresse, dans toutes les activités. Celles-ci peuvent donc avoir accès à tous les objets du site. Des contrôles limitant l'accès ne peuvent être réalisés que de manière partielle, les protections offertes par le système de mémoire partagée d'Unix n'étant plus utilisables.

Les opérations de la gestion des blocs de mémoire partagée sont les suivantes :

- Allocation d'un bloc de mémoire partagée à un objet pour la création du segment de son image.
- Libération du bloc de mémoire partagée alloué à un objet, au moment de la destruction de l'image de celui-ci.
- Attachement du segment de mémoire partagée à la même adresse virtuelle dans toutes les activités.
- Détachement du segment de mémoire partagée.

6.1.2.2 Version multi-segments de la gestion de la mémoire partagée

Le segment d'une image est implanté par un segment de mémoire partagée Unix. Ainsi, lors de son verrouillage sur un site, tout objet se voit allouer un segment de mémoire partagée pour la mise en oeuvre de son image. La taille de ce segment est exactement celle de l'état de l'objet ou légèrement supérieure, compte tenu que la taille d'un segment de mémoire partagée doit être un multiple de la taille minimale d'allocation. Le segment reste alloué jusqu'à la destruction de l'image.

Puisque l'allocation et la libération des segments se fait en utilisant directement les primitives Unix de gestion de mémoire partagée, la fonction du module se restreint à la gestion des adresses virtuelles auxquelles les activités doivent attacher ces segments. Contrairement à la version mono-segment, un segment partagé par plusieurs activités peut être attaché à des adresses virtuelles différentes dans chacune d'elles.

Les opérations de la gestion des segments de mémoire partagée sont les suivantes :

- Allocation d'un segment de mémoire partagée à un objet pour la création du segment de son image.
- Détermination de l'adresse virtuelle d'attachement d'un segment de mémoire partagée dans une activité.
- Attachement d'un segment de mémoire partagée à une adresse virtuelle d'une activité.
- Détachement d'un segment de mémoire partagée.
- Libération de l'adresse virtuelle d'une activité à laquelle était attaché un segment de mémoire partagée.
- Libération du segment de mémoire partagée alloué à un objet, au moment de la destruction de l'image de clui-ci.

6.1.3 Support à la réalisation d'objets stockés

Un volume est implanté par une partition-disque Unix à laquelle on accède en mode *raw*. Un bloc de système d'objets correspond donc à un bloc-disque Unix. A l'intérieur de ces partitions, les objets stockés sont réalisés par un ensemble de blocs-disque organisés en arborescence.

Pour que les activités d'un site puissent accéder facilement aux blocs d'un objet stocké tout en minimisant les entrées/sorties-disque, deux modules ont été superposés à l'interface-disque en mode *raw* d'Unix. Il s'agit du module de la gestion du cache et du module de la gestion des blocs.

Le module de la gestion du cache met en oeuvre un cache de blocs de système d'objets. Il s'agit d'un cache très simplifié par rapport à celui du système de fichiers d'Unix mais qui s'adapte aux besoins d'accès aux objets stockés. Il existe un cache (et un seul) sur chaque site dont la mémoire sécondaire est dédiée à la réalisation d'un ou plusieurs systèmes d'objets.

Le module de gestion des blocs effectue la gestion de l'espace-disque des systèmes d'objets mis en oeuvre dans la mémoire secondaire d'un site. Ce module alloue et libère les blocs.

Dans la première version de Guide, les systèmes d'objets sont mono-volumes. Sauf indication du contraire, les termes 'volume' et 'système d'objets' sont désormais équivalents.

6.1.3.1 Gestion du cache

A tout moment, le cache contient les blocs les plus récemment utilisés. Quand l'accès à un bloc est demandé, celui-ci est d'abord recherché dans le cache et il n'est transféré depuis le disque qu'en cas d'échec. De même, l'écriture des blocs est réalisé de façon asynchrone et est retardée au maximum. Le cache permet ainsi de minimiser les échanges entre les mémoires principale et secondaire.

Le cache d'un site est constitué d'un ensemble de tampons, dont le nombre est un paramètre de génération du système. Chaque tampon contient un bloc d'un système d'objets[16]. L'allocation ou la réallocation d'un tampon se font lors d'une demande d'accès à un bloc non présent dans le cache. Ce tampon restera alloué à ce bloc jusqu'à ce que, pour des raisons de manque d'espace, il soit alloué à un nouveau bloc.

Lors de son allocation à un bloc, un tampon n'est pas immédiatement chargé avec le contenu de ce bloc. En effet, celui-ci peut ne pas être significatif. Le chargement, si

16. Le terme 'tampon' est utilisé à la place de ceux de 'bloc' ou de 'bloc de cache' afin d'éviter des confusions avec le terme 'bloc d'un système d'objets'.

nécessaire, doit donc être explicitement effectué après verrouillage du tampon (cf. ci-après).

Les opérations de la gestion du cache sont les suivantes :

- Allocation d'un tampon. Cette opération stocke préalablement le bloc contenu dans le tampon si celui-ci a été modifié.
- Verrouillage d'un tampon. Cette opération permet l'accès exclusif à un tampon alloué à un bloc déterminé. Elle fait appel à l'opération d'allocation si le bloc n'est pas présent dans le cache.
- Déverrouillage d'un tampon.
- Lecture d'un bloc dans un tampon.
- Ecriture (stockage) du bloc contenu dans un tampon.
- Sauvegarde asynchrone des blocs modifiés du cache. Cette opération écrit tous les blocs modifiés contenus dans les tampons non verrouillés.

6.1.3.2 Gestion des blocs d'un système d'objets

Ce module gère les blocs-disque de la partition Unix supportant un système d'objets. Ses fonctions principales sont la réservation et la libération des blocs. Le module réalise également l'allocation des estampilles contenues dans les oid's des objets créés dans ce système d'objets (cf. section 4.2).

Selon leur contenu, les blocs d'un système d'objets peuvent appartenir à quatre catégories :

1. Les blocs utilisés pour la gestion du système d'objets. Il s'agit, d'une part, du bloc qui contient les informations générales de gestion et, de l'autre, des blocs qui contiennent les adresses des blocs libres du système d'objets.
2. Les blocs de descripteur. Ce sont les blocs qui contiennent les descripteurs en MPO des objets stockés rangés dans le système d'objets.
3. Les blocs de données. Ce sont les blocs qui contiennent l'état des objets stockés rangés dans le système d'objets. Les blocs de données peuvent contenir des adresses d'autres blocs de données : ce sont ces 'blocs d'indirection' qui permettent de mettre en oeuvre l'arborescence de blocs d'un objet stocké.
4. Les blocs libres.

Les opérations de la gestion des blocs d'un système d'objets sont les suivantes :

- Obtention d'une estampille d'objet.
- Réservation d'un bloc de descripteur.

- Réservation d'un ou plusieurs blocs de données.
- Libération d'un bloc de descripteur.
- Libération d'un bloc de données.

6.2 Principes de la réalisation répartie du modèle d'exécution

Une activité qui exécute une méthode d'un objet dont l'image a été créée sur un site est mise en oeuvre par un processus Unix qui s'exécute dans ce site et qui attache, dans sa zone partagée, le segment de l'image de l'objet.

Lorsqu'une activité appelle une méthode d'un objet verrouillé sur un site autre que celui où elle s'exécute, la gestion de domaines et d'activités met en oeuvre le mécanisme suivant :

- Recherche du site de verrouillage de l'objet (cf. sections 4.3, où la fonction de recherche reçut précisement le nom de 'fonction de verrouillage', et 6.3.3). Soit le site *x*.
- Extension du domaine de l'activité sur le site *x*. Cette opération est appelée **diffusion** du domaine et consiste à créer sur *x* les structures de données nécessaires pour la représentation du domaine si elles n'y existent pas déjà.
- Création sur *x* de l'image de l'objet si elle n'y existe pas déjà.
- Liaison de l'objet dans le domaine.
- Création sur *x* d'un processus qui exécute la méthode de l'objet pour le compte de l'activité appelante. Le segment de l'image de l'objet doit donc être attaché dans la zone partagée du processus créé. Ce processus constitue l'**extension** sur *x* de l'activité appelante.

Le mécanisme d'appel à un objet distant n'est pas visible pour les applications programmées en langage Guide. En effet, pour elles, il n'existe pas de différence entre les appels locaux et les appels distants. La diffusion d'un domaine et l'extension d'une activité sont montrées dans la figure 6.3.

Répartition des domaines et des activités

Un domaine est créé sur un site. Il peut ensuite devenir multi-sites par le biais du mécanisme de diffusion. Dans chaque site où il existe, un domaine possède un représentant local, dit **domaine local**. Chaque domaine local à un site est pourvu d'un **descriptif local** dont la tâche est de grouper tous les objets liés dans le domaine et possédant une image sur le site.

Le domaine local au site de création est le domaine principal. Les domaines locaux créés par diffusion sont des domaines secondaires. Un domaine ne 'diffuse' pas sur un site où il y a déjà un représentant local.

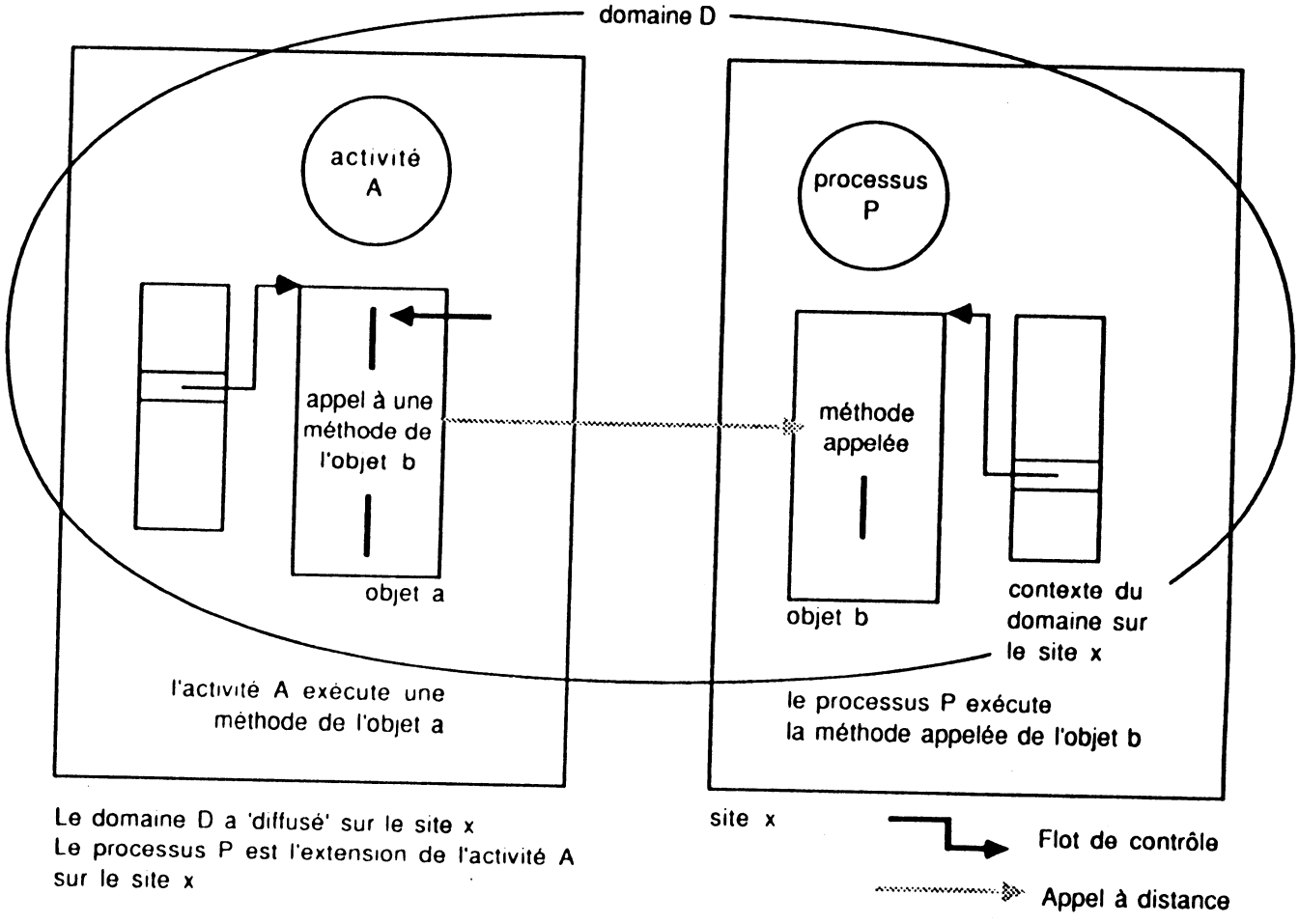


Figure 6.3 - Accès à un objet distant

Une activité d'un domaine est créée sur un site. Elle peut ensuite devenir multi-sites par le biais du mécanisme d'extension. Une activité est donc composée d'un ensemble d'extensions, chacune dans un domaine local. Une activité n'a nécessairement pas une extension dans chaque représentant local de son domaine.

L'extension d'une activité qui s'exécute dans le site de création de l'activité est appelée activité principale. Les autres extensions sont des activités secondaires. Une activité ne s'étend pas sur un site où il y a déjà une extension : La méthode en question sera exécutée sur l'objet distant par l'extension existante.

Réalisation des processeurs virtuels de la machine à objets

On rappelle qu'un processeur virtuel est une 'machine' qui interprète des appels de méthode. Dans la mise en oeuvre du modèle d'exécution, un tel processeur correspond exactement à l'ensemble des extensions d'une activité.

L'allocation d'un processeur virtuel consiste, d'une part, en la création de l'activité principale de cet ensemble et, de l'autre, en l'attachement, à l'intérieur de sa zone partagée, du segment de l'image de l'objet sur lequel s'exécute la méthode.

Puisque les extensions sont créées par un mécanisme d'appel procédural à distance, une seule de ces extensions est active à un instant donné. L'objet dont l'une des méthodes est exécutée par l'extension active n'est autre que l'objet courant du processeur virtuel.

Etant donné qu'une seule de ces extensions est active à la fois, deux extensions qui se 'synchronisent' sur le même site appartiennent forcément à des ensembles d'exécutions différents. De cette manière a lieu la synchronisation de processeurs virtuels.

Domaines locaux

Afin de maintenir la cohérence entre les activités principales et secondaires d'un domaine local à un site, il est nécessaire de forcer les processus Unix qui les implantent à avoir le même espace virtuel. Ceci a été réalisé de la manière suivante :

- Le descriptif local du domaine local est réalisé par un segment de mémoire partagée que toutes les activites (i.e., les processus qui les réalisent) du domaine local attachent à la même adresse virtuelle.
- Si deux ou plusieurs activités veulent exécuter des méthodes d'un objet lié dans le domaine, elles doivent attacher le segment de son image à la même adresse virtuelle.

Dans le cas où la gestion de la mémoire se fait en version mono-segment, toutes les activités attachent ce seul segment de mémoire partagée à la même adresse.

Dans le cas où chaque segment d'image est implanté par un segment de mémoire partagée, les deux considérations suivantes doivent être prises en compte pour son attachement :

1. L'opération qui détermine son adresse d'attachement dans une activité n'est exécutée que par la première des activités qui l'attachent. Les activités qui veulent l'attacher par la suite doivent le faire *à cette adresse*. En conséquence, l'opération qui invalide l'adresse ne peut être exécutée que par la dernière des activités qui détachent le segment.
2. L'on maintient dans un seul segment de mémoire partagée le descriptif local et les structures de données de gestion des adresses d'attachement. Ceci permet à toutes les activités de partager les mêmes structures et d'attacher les segments toujours à la même adresse.

Bien évidemment, les activités des autres domaines locaux au site peuvent attacher les segment à des adresses différentes.

Dorénavant, lorsqu'on parle d'une activité qui attache ou détache un segment d'image, il est sous-entendu que la gestion de la mémoire est faite en version multi-segments.

Le descriptif d'un domaine

Le descriptif local d'un domaine local à un site comporte une entrée pour chaque objet possédant les caractéristiques suivantes :

- Il est verrouillé sur le site.
- Son image a été effectivement créée dans un segment partagée de la mémoire virtuelle du site.
- Il est lié dans le domaine.

Une entrée du descriptif local comporte donc :

- La position du descripteur en MVO de l'objet (cf. section 6.3.4.1)[17].
- L'adresse virtuelle d'attachement, dans chaque activité, du segment de son image.
- Le nombre d'activités ayant attaché ce segment.

Le démon Guide

Les primitives de la gestion de domaines et activités sont réalisées sur un site par un processus Unix présent en permanence, appelé le **démon** Guide. Il remplit deux fonctions de base : D'une part, il gère les données des domaines locaux et des activités du site et, de l'autre, il se charge du traitement des requêtes externes concernant les domaines ou activités représentés (ou qui doivent s'étendre) sur le site.

Le démon est en fait le processus Unix le plus important d'un site Guide. Il démarre le système dans le site, crée et initialise toutes les structures de données du site, crée et fait diffuser les domaines et lance (*fork*) les activités et les autres processus du site. Lors de la création ou diffusion d'un domaine sur le site, il crée le segment de mémoire partagée qui réalise le descriptif local du domaine sur le site.

Avant la fin de son exécution, le démon libère toutes ses structures de donnés et détruit tous les domaines locaux et les processus du site.

---

17. Dans la version actuelle de Guide, le maintien de cette position n'est pas nécessaire. Ceci est du au fait que la table qui regroupe les descripteurs en MVO et tous les descriptifs locaux de tous le domaines locaux ont le même nombre d'entrées.

Exécution d'une méthode d'objet par une activité

Lorsqu'une activité principale ou secondaire d'un domaine local à un site veut exécuter l'une des méthodes d'un objet, les actions suivantes ont lieu :

1. Si une image de l'objet a été créée sur le site :
   - Si l'objet est verrouillé sur le site mais n'est lié dans aucun domaine ayant un représentant sur le site, alors il est lié dans le domaine de l'activité.
   - Si l'objet est verrouillé sur le site et s'il est lié dans un autre domaine ayant un représentant sur le site, alors il est lié dans le domaine de l'activité.
   - Si l'objet n'est pas verrouillé sur le site (par conséquent, il ne peut être lié dans aucun domaine ayant un représentant sur le site), son image doit être validée. Dans ce cas :
     — Si l'image est valide, l'objet est à nouveau verrouillé sur le site. Il est ensuite lié dans le domaine de l'activité.
     — Si l'image n'est pas valide, elle est détruite. On se rapporte ensuite au cas 2.
2. Si aucune image de l'objet n'est présente sur le site, la gestion de domaines et d'activités demande au niveau bas de la machine virtuelle de verrouiller l'objet sur le site. Dans ce cas :
   - Si l'objet est déjà verrouillé sur un autre site, la demande de verrouillage est refusée et l'identificateur de cet autre site est retourné. Le domaine de l'activité doit alors 'diffuser' sur ce site (s'il n'y est pas déjà présent) et l'activité doit y créer une extension (si elle n'y existe déjà). Si l'image n'a pas été créée sur ce site, elle y est créée. Ensuite, l'objet est lié dans le domaine de l'activité.
   - Si l'objet n'est verrouillé sur aucun site, il est verrouillé sur le site ayant émis la demande. Après la création de son image sur ce site, l'objet est lié dans le domaine de l'activité.
3. Si une image de la classe de l'objet n'est pas présente sur le site de verrouillage de l'objet, elle y est créée. La classe est liée dans le domaine de l'activité.
4. Afin de lier un objet et sa classe dans le domaine d'une activité, deux nouvelles entrées sont ajoutées dans le descriptif local du représentant du domaine dans le site de verrouillage de l'objet.
5. L'activité attache ensuite le segment de l'image de l'objet et le segment de l'image de la classe aux adresses indiquées dans le descriptif local.
6. La méthode est exécutée.
7. Si l'activité n'exécute plus de méthodes sur l'objet, le segment de son image et le segment de l'image de la classe sont détachés.

8. Si l'activité n'exécute plus de méthodes sur l'objet et si plus aucune activité ne s'exécute sur l'objet, celui-ci et sa classe sont déliés du domaine de l'activité.

Pour réaliser les cinq premières actions, l'activité exécute la primitive `mvo_linkAsObject`. Pour réaliser les deux dernières, elle exécute la primitive `mvo_unLinkAsObject`. Il s'agit des deux primitives de la gestion de domaines et d'activités qui déclenchent l'exécution des primitives de la gestion de la mémoire d'objets (cf. section 6.3).

La suite d'actions suivante :

- Appel à la primitive `mvo_linkAsObject`,
- sauvegarde du contexte d'exécution courant,
- appel effectif à la méthode,
- appel à la primitive `mvo_unLinkAsObject` et
- restitution du contexte d'exécution,

constitue le corps de la primitive `guideCall`, qui fait partie de l'interface de la machine virtuelle. Tout appel de méthode d'objet en langage Guide est traduit par le compilateur dans un appel à la primitive `guideCall`.

L'activité peut aussi exécuter un `guideCreate`, une autre primitive de l'interface de la machine virtuelle. Tout appel à la méthode `New` sur une classe est traduit par le compilateur dans un appel à la primitive `guideCreate`. Avant la création de l'objet stocké correspondant à une instance d'une classe, l'activité doit lier cette classe dans son domaine et attacher le segment de son image dans sa zone partagée. Les actions contraires ont lieu avant la fin de la primitive.

Les primitives `mvo_linkAsObject` et `mvo_unLinkAsObject` jouent un rôle essentiel dans la réalisation de la machine virtuelle. D'une part, leur exécution déclenche l'exécution des primitives de la gestion de la mémoire d'objets. De l'autre, elles sont appelées par les primitives `guideCall` et `guideCreate`. En conséquence, elles constituent *le pont* entre l'exécution d'une méthode, objectif du modèle d'exécution, et la gestion de la mémoire d'objets, objectif du modèle de la mémoire.

6.3 Réalisation du modèle de la mémoire

La mise en oeuvre du modèle de la mémoire définit la gestion d'images et la gestion d'objets stockés. Ces gestions réalisent, en étroite collaboration, les fonctions de gestion de la mémoire d'objets. Ces fonctions ont été identifiées dans la section 4.1.3 ; on les rappelle ici :

1. Création d'une image.
2. Destruction d'une image.

3. Création d'un objet stocké.
4. Destruction d'un objet stocké.
5. Validation d'une image.
6. Chargement en MVO d'un objet stocké.
7. Rangement en MPO d'une image.
8. Verrouillage d'un objet.
9. Déverrouillage d'un objet.

La gestion d'images réalise les fonctions de création et de destruction d'une image. La gestion d'objets stockés effectue les fonctions de création et destruction d'un objet stocké ainsi que celles de chargement en MVO et de rangement en MPO. La réalisation des trois fonctions restantes se complique du fait qu'un objet peut posséder plusieurs images. On verra plus tard qui se charge de les mettre en oeuvre (cf. section 6.3.3).

6.3.1 Principes de la réalisation de la gestion d'images

Une mémoire virtuelle de la machine à objets a été définie comme la mémoire virtuelle dont un processeur virtuel dispose pour la définition des objets qu'il peut adresser. Puisqu'un processeur virtuel correspond à l'ensemble des extensions d'une activité (cf. section 6.2), une mémoire virtuelle est réalisée par l'union des espaces d'exécution de toutes les extensions correspondantes au processeur.

Par conséquent, une mémoire virtuelle de la machine à objets consiste en un ensemble d'images distribué sur les sites visités par les extensions. Elle est donc une entité multi-sites.

Etant donné que les images des entités sont mono-sites et que le verrouillage des objets se fait sur un site, les fonctions que la gestion d'images réalise sur un site doivent se restreindre aux images du site. La gestion d'images n'est donc pas répartie sur l'ensemble des sites Guide mais 'relative' à chaque site. La gestion d'images est la même sur chaque site mais elle est complètement indépendante de celle qui a lieu sur un autre site.

Une mémoire virtuelle de la machine à objets s'étalant sur plusieurs sites, la gestion des images qu'elle comporte ne peut pas être comprise dans la gestion d'images d'un seul site. La mémoire virtuelle d'un processeur virtuel est découpée en partitions de façon à ce que chaque partition comporte les images créées sur un site différent. Ainsi, la gestion des images de la partition correspondante au site x sera comprise dans la gestion des images de x.

En conséquence, le terme 'mémoire virtuelle de la machine à objets' n'a pas de sens dans la réalisation de la gestion d'images. On se rapporte plutôt au terme 'MVO d'un site'. On appelle **MVO d'un site** la mémoire virtuelle du site, plus exactement la partie de cette mémoire réservée pour la création d'images, augmentée des fonctions de

gestion de segments (l'une des deux versions) et des fonctions de la gestion d'images relative au site. La MVO de Guide est réalisée par l'union des MVOs des sites Guide.

6.3.2 Principes de la réalisation de la gestion d'objets stockés

Un système d'objets est **local à un site** lorsque l'un des volumes qui le composent est implanté sur l'un des supports de mémoire secondaire du site. On appelle **MPO d'un site** l'union des systèmes d'objets locaux au site. La MPO de Guide est réalisée par l'union des MPOs des sites Guide.

Un site est le **site de résidence d'un système d'objets** si sur l'un de ses supports de mémoire secondaire est implanté le volume principal du système d'objets. Dans la version actuelle de Guide, un système d'objets ne comporte qu'un seul volume. Par conséquent, son site de résidence est le seul auquel un système d'objets est local. Un cache n'est implanté que sur le site de résidence d'un ou plusieurs systèmes d'objets.

Un objet stocké est **local à un site** s'il est rangé dans l'un des systèmes d'objets locaux au site. Cependant, le **site de résidence d'un objet stocké** est celui où réside son système d'objets 'de résidence' (cf. section 4.2).

La figure 6.4 montre les sites de résidence d'un système d'objets et d'un objet stocké.

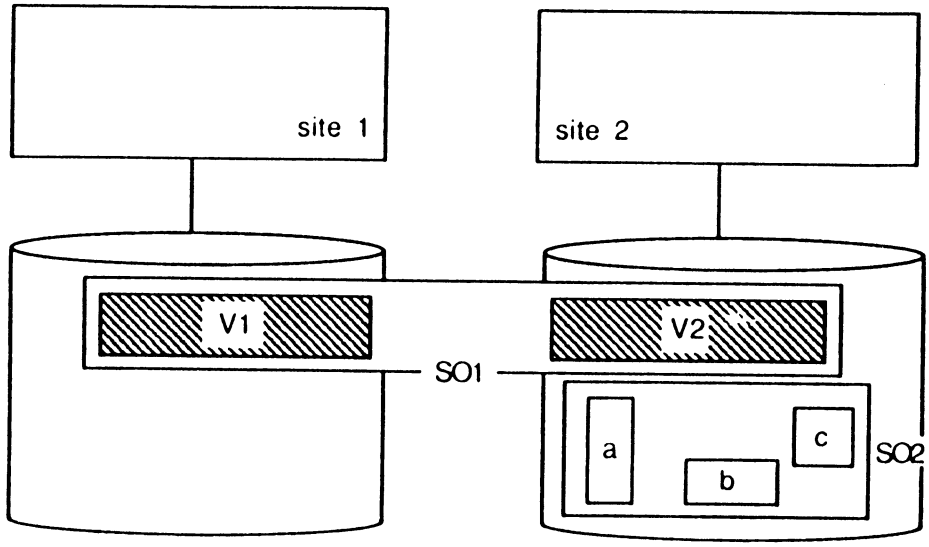
Contrairement à la gestion d'images, les fonctions que la gestion d'objets stockés réalise sur un site ne se restreignent pas aux objets stockés qui résident dans le site. En effet, bien qu'un objet stocké ne réside que dans un site, il peut être chargé dans la mémoire virtuelle d'un autre pour la création d'une image. De même, le rangement en MPO d'une image se réalise à partir du site où cette image se trouve, même si l'objet stocké correspondant réside dans un site différent.

En fait, *toutes* les fonctions de gestion d'objets stockés peuvent être réalisées sur des sites différents de ceux où résident les objets stockés. La gestion d'objets stockés est effectivement répartie sur l'ensemble des sites Guide. Les primitives de gestion d'objets stockés peuvent donc être exécutées par une activité sur un site quelconque, même s'il est dépourvu de mémoire secondaire ou si celle-ci n'est pas utilisée pour implanter des systèmes d'objets.

Le mécanisme d'accès aux blocs d'un objet stocké distant a besoin des supports suivants :

1. Les processus-serveurs : Un certain nombre de processus-**serveurs** est créé sur chaque site de résidence d'un système d'objets. Le nombre de ces processus est un paramètre de configuration du système. Un processus-serveur d'un site x permet aux activités distantes d'accéder aux blocs des objets stockés qui résident dans x.

2. La gestion d'objets stockés 'rélative' à chaque site, c'est-à-dire, restreinte aux objets stockés résidant dans un site. Il s'agit de l'ensemble de primitives qui, sur chaque site, réalisent effectivement les fonctions de gestion d'objets. Cette



V1 est le volume principal de SO1
En conséquence, site 1 est le site de résidence de SO1
SO2 réside dans site 2
SO2 est le système d'objets de résidence des objets stockés a, b et c
En conséquence, site 2 est le site de résidence de a, b et c

Figure 6.4 - Sites de résidence d'un système d'objets et d'un objet stocké

gestion est la même sur chaque site mais elle est complètement indépendante de celle qui a lieu sur un autre site. A chaque primitive de gestion répartie d'objets stockés correspond une primitive de gestion d'objets stockés dans son 'sens relatif'.

Le mécanisme d'accès distant à un objet stocké, illustré par la figure 6.5, est mis en oeuvre de la manière suivante :

Considérons une activité qui s'exécute sur le site *x*. Lorsqu'elle veut accéder aux blocs d'un objet stocké, elle exécute sur *x* la primitive de la gestion répartie nécessaire pour avoir l'accès. La première action d'une telle primitive est déterminer si l'objet stocké concerné réside dans le site d'exécution de l'activité (*x*) ou s'il est distant.

L'indicateur de localisation en MPO, contenu dans la référence-système de l'objet correspondant (cf. section 4.2), indique à l'activité le système d'objets de résidence. Ensuite, les informations de configuration du système Guide lui permettent de connaître le site de résidence de l'objet stocké.

Si l'objet stocké réside dans *x*, la primitive fait appel à la primitive correspondante de la gestion relative au site *x*. L'exécution effective de la fonction de gestion permet à l'activité d'accéder directement aux blocs de l'objet stocké.

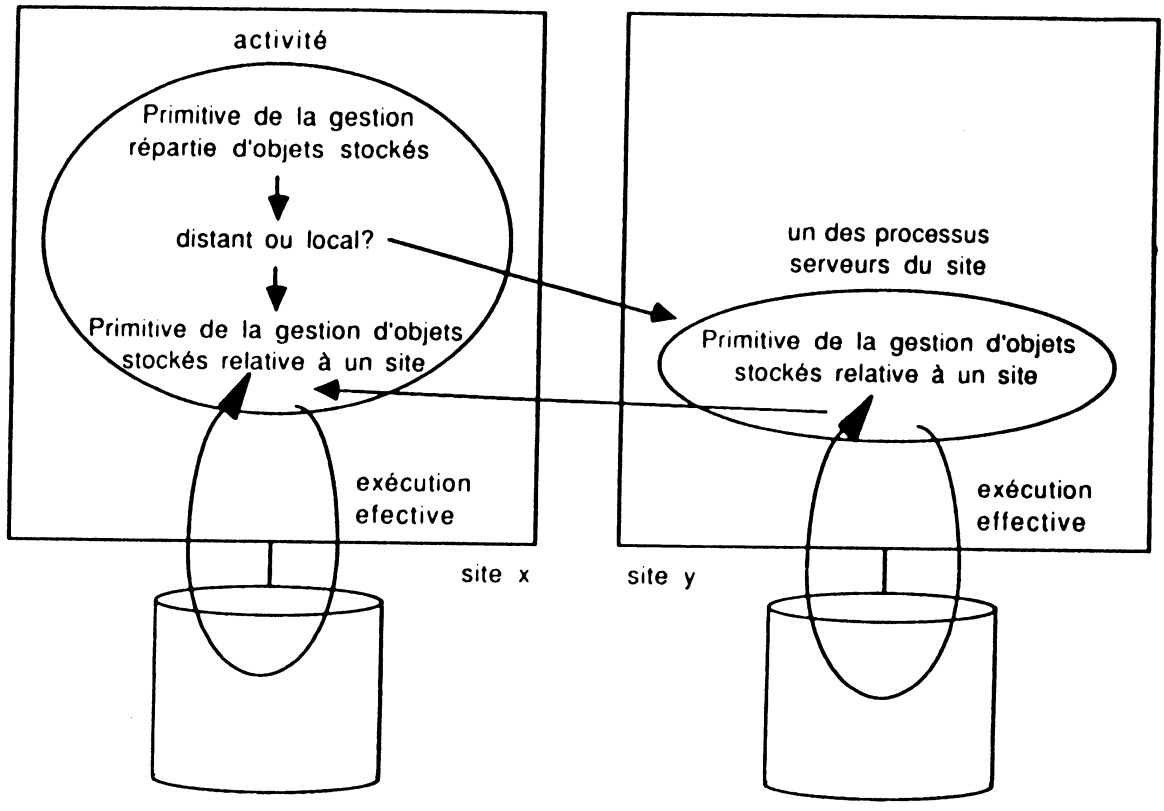


Figure 6.5 - Accès distant à un objet stocké

Si l'objet stocké réside dans un autre site, disons *y*, l'activité doit envoyer à ce site une requête comportant l'identificateur de son site d'exécution (*x*), le nom de la primitive et tous ses paramètres. L'activité se met ensuite en attente de la réponse.

L'un des processus-serveurs du site *y* reçoit la requête et exécute la primitive correspondante de la gestion relative au site *y*. L'exécution effective de la fonction de gestion permet au processus-serveur d'accéder directement aux blocs de l'objet stocké. A la fin de l'exécution de la primitive, le processus-serveur envoie les résultats à l'activité. Une fois la réponse reçue, l'activité continue son exécution normale. Quant au processus-serveur, il se met en attente d'une nouvelle requête.

Lorsqu'une activité exécute la primitive de chargement d'un objet stocké et que celui-ci est distant, le processus-serveur qui réalise effectivement la fonction envoie à l'activité tous les blocs pour qu'elle puisse en créer l'image sur son site d'exécution. Dans le cas du rangement en MPO d'une image, l'activité l'envoie au processus-serveur pour que celui-ci la range sur les blocs de l'objet stocké correspondant.

Les primitives de la gestion relative à un site sont décrites dans la section 6.3.5. Les primitives de la gestion répartie sont décrites dans la section 6.3.6. Il est important de souligner que l'identificateur du site d'exécution de l'activité qui émet une requête est

indispensable pour le bon déroulement des primitives de la gestion relative à un site (cf. sections 6.3.3 et 6.3.5).

6.3.3 Verrouillage, déverrouillage et validation

La réalisation des fonctions de verrouillage, déverrouillage et validation s'avère compliquée à cause de la faculté des objets de posséder plusieurs images sur des sites différents. Cependant, puisqu'un objet stocké ne possède qu'un seul site de résidence, il a été choisi de faire mettre en oeuvre ces trois fonctions par la gestion d'objets stockés plutôt que par la gestion d'images. Ceci est d'autant plus nécessaire que la gestion d'images a été restreinte aux images créées sur un site.

Ainsi, pour un objet, son verrouillage, son déverrouillage et la validation de ses images sont mises en oeuvre par la gestion d'objets stockés relative au site de résidence de l'objet stocké correspondant. Sur ce site, ces fonctions peuvent être exécutées par une activité ou par un processus-serveurs à la demande d'une activité distante.

Le descripteur d'un objet en MPO comporte donc des informations qui permettent de verrouiller l'objet, de le déverrouiller et de valider ses images. Il s'agit des trois variables suivantes :

1. `NextSite`, qui contient l'identificateur du site qui a verrouillé l'objet mais qui n'a pas encore chargé l'objet stocké correspondant[18].
2. `OwnerSite`, qui contient l'identificateur du site ayant verrouillé l'objet et ayant chargé l'objet stocké correspondant. Il s'agit du site où se trouve la seule image valide de l'objet.
3. `LastSite`, qui contient l'identificateur du dernier site ayant créé, (éventuellement) modifié et rangé en MPO l'image de l'objet.

La constante `NOSITE` est utilisée pour donner une valeur invalide à ces variables.

Verrouillage et déverrouillage d'un objet

Le verrouillage d'un objet est mis en oeuvre en modifiant de la manière suivante les variables `OwnerSite` et `NextSite` de son descripteur :

- A la demande de verrouillage de l'objet par un site, la variable `NextSite` est mise à jour avec l'identificateur du site. Dans ce cas :

---

18. Lorsqu'on dit qu''un site exécute (ou réalise) telle ou telle fonction' on veut dire effectivement qu'il y a, sur ce site, une activité qui exécute (ou réalise) la fonction. Vue l'utilisation que l'on fait des identificateurs des sites dans cette section, il nous a paru qu'un tel abus de langage rend la description plus claire.

— Toute demande de verrouillage de l'objet ou de chargement de son objet stocké provenante d'un site autre que celui indiqué par `NextSite` est refusée.

— Si le site identifié par `NextSite` demande de verrouiller l'objet, le gestion d'objets stockés lui indique que cette opération a déjà été effectuée.

— Si le site identifié par `NextSite` demande de charger l'objet stocké, celui-ci est chargé et les variables `OwnerSite` et `NextSite` sont mises à jour : La première avec l'identificateur du site ayant effectué le chargement et la deuxième avec `NOSITE`.

- Lorsque l'objet est verrouillé et son objet stocké chargé, toute nouvelle demande de verrouillage ou de chargement est refusée.
- Les variables `NextSite` et `OwnerSite` du descripteur d'une classe ne sont jamais mises à jour. Leur valeur est toujours `NOSITE`. Ceci permet de ne jamais verrouiller les classes.

Lorsqu'un site déverrouille un objet, la variable `LastSite` est mise à jour avec l'identificateur du site. La variable `OwnerSite` est mise à jour avec `NOSITE` afin d'indiquer que le site n'utilise plus l'image de l'objet. Remarquons que ceci n'implique nullement que l'image a été détruite.

Validation d'une image d'objet

La variable `LastSite` est utilisée pour la validation des images d'un objet. Deux cas se présentent :

- Toute demande de validation provenant d'un site autre que celui indiqué par `LastSite` est refusée. L'objet est verrouillé mais son objet stocké correspondant doit être chargé à nouveau.
- Si la demande de validation provient du site identifié par `LastSite`, l'image est valide. En effet, aucun site n'a créé une image de l'objet depuis que celui indiqué par `LastSite` l'a déverrouillé. L'objet est verrouillé et la gestion d'objets stockés considère qu'il est déjà chargé. La variable `OwnerSite` est mise à jour avec l'identificateur du site ayant validé l'image.

Les transitions d'état pour un objet

Reprenant les idées de la figure 4.9, les différents états dans lesquels un objet peut se trouver au sein de la mémoire d'objets sont les suivants :

- Non verrouillé et non chargé : Un objet est dans cet état lorsqu'il ne possède pas d'image sur aucun des sites.
- Verrouillé et non chargé : Un objet est dans cet état s'il ne possède et ne peut posséder qu'une seule image et que celle-ci est en en cours de création par

chargement de l'objet stocké correspondant.

- Verrouillé et chargé : Un objet est dans cet état s'il comporte une image valide sur un site. Il peut ou non posséder d'autres images sur d'autres sites.
- Non verrouillé et chargé : Un objet est dans cet état lorsqu'il comporte une ou plusieurs images sur différents sites et qu'aucune n'est valide.

Du point de vue de la MPO, l'état 'non verrouillé et chargé' est équivalent à l'état 'non verrouillé et non chargé' car, l'objet ayant été déverrouillé, il peut à nouveau être verrouillé sur un autre site pour qu'une image valide y soit créée.

Un raffinement de la figure 4.9, motivé par l'introduction des variables `NextSite`, `OwnerSite` et `LastSite`, est présenté dans la figure 6.6. Cette figure montre les différentes transitions possibles pour l'état dans lequel se trouve un objet lors des interactions entre la MVO d'un site *x* et la MPO.

Sur cette figure apparaissent des transitions d'état qui n'ont pas de correspondant dans la figure 4.9. C'est le cas des transitions 7 (validation réfusée), 8 (verrouillage suivi de chargement) et 9 (déverrouillage réalisé par un autre site). La transition 8 résulte d'une optimisation : L'objet stocké étant vide (ses blocs sont réservés mais pas encore initialisés), il n'y a rien à charger en MVO. Une fois l'objet verrouillé, son objet stocké est considéré comme étant chargé.

6.3.4 Gestion d'images relative à un site

La gestion d'images d'un site consiste en la création et en la destruction d'images d'objet sur le site. Ces fonctions concernent aussi bien les descripteurs en MVO que les segments des images. La gestion de ces segments est réalisée par les modules de gestion de la mémoire partagée (versions mono- et multi-segments).

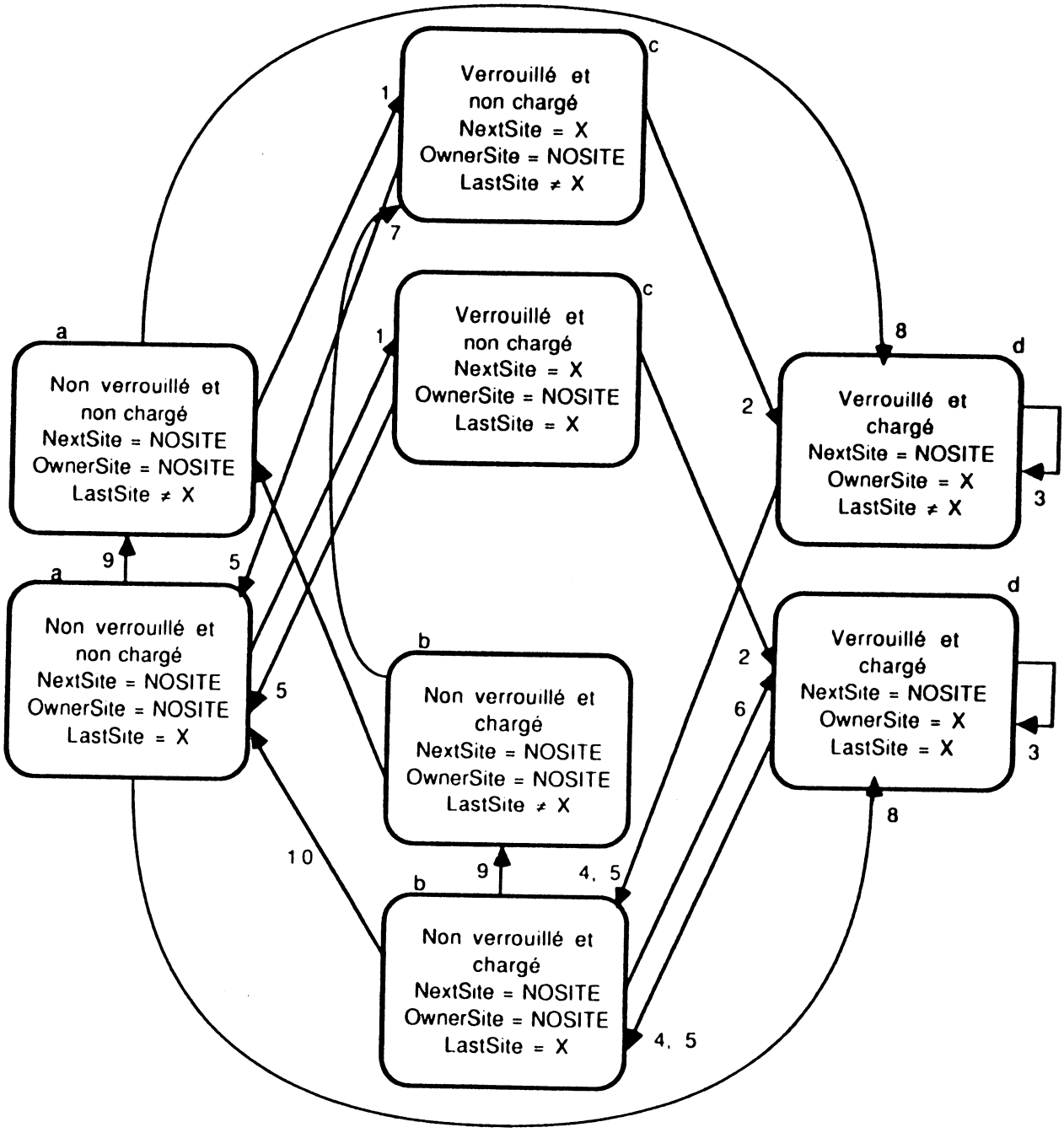
Sur un site, les descripteurs en MVO sont regroupés dans une table unique nommée **Table de la Mémoire Virtuelle d'Objets** (TMVO). Cette table comporte donc toutes les informations de gestion sur l'ensemble des images créées sur le site.

Les références d'exécution comportent un indicateur de localisation dans la MVO d'un site (cf. section 4.2). Pour un objet, la valeur de cet indicateur est la position (index) que son descripteur en MVO occupe à l'intérieur de la TMVO du site.

Les opérations du module de la gestion d'images relative à un site sont donc :

- Création d'une image sur le site.
- Destruction d'une image.

Ces opérations sont mises en oeuvre non seulement par les primitives du module mais aussi par un ensemble d'opérations internes qui seront décrites dans les sections suivantes.



Etats pour un objet :
cf. figure 4.9

Transitions :
cf. figure 4.9

7. Validation refusée
8. Verrouillage + chargement
9. Verrouillage, chargement, rangement éventuel et déverrouillage par un site autre que X

Figure 6.6 - Transitions d'états pour un objet

6.3.4.1 Description de la TMVO

Une entrée de la TMVO est allouée à un objet lorsque celui-ci est verrouillé. Elle redevient disponible dès que l'image de l'objet est détruite. Une fonction de dispersion (*hash code*) a été implantée pour accélérer l'accès aux descripteurs en MVO.

Les opérations de gestion de la TMVO sont les suivantes :

- Recherche d'une entrée.
- Allocation d'une entrée.
- Libération d'une entrée.

Ces opérations sont internes au module de la gestion d'images relative à un site. Elles sont appelées par les opérations de création et destruction d'images. L'accès à chaque entrée de la TMVO s'effectue en exclusion mutuelle puisqu'elle risque d'être modifiée par chaque activité s'exécutant sur le site.

Un segment de mémoire partagée Unix de taille fixe est créé pour maintenir les entrées de la TMVO et les valeurs des variables de gestion de cette table.

6.3.4.2 Création, modification et destruction d'images

La création d'une image se réalise en quatre étapes :

1. Allocation d'une entrée dans la TMVO pour le descripteur en MVO.
2. Initialisation de ce descripteur à partir des informations du descripteur en MPO.
3. Création d'un segment ou réservation d'un bloc de segment (dans le cas de la version mono-segment) pour le segment de l'image.
4. Initilisation de ce segment ou bloc en chargeant l'objet stocké correspondant.

Les trois premières étapes sont réalisées par une opération interne au module de gestion d'images : `loadDescr`. L'appel à la primitive `MPO_GetDesc` (cf. section 6.3.6) permet d'obtenir les informations du descripteur en MPO et de verrouiller l'objet ou de déterminer le site où il est déjà verrouillé.

La dernière étape est réalisée par la primitive `mvo_loadObject` (cf. section 6.3.4.3).

La référence-système d'un objet et sa position dans la TMVO sont les deux informations qui constituent sa référence d'exécution (cf. section 4.2). Or, bien que ces deux informations existent et qu'elles soient utilisées amplement par la gestion d'images, elles ne sont jamais fusionnées dans une structure de données unique. Dans la première version de Guide, les références d'exécution n'ont pas été mises en oeuvre.

La destruction d'images a lieu lors de la récupération d'espace dans la MVO du site. Dans cette version du système, où aucun ramasse-miettes n'a été réalisé, la

récupération est réalisée par une opération interne au module de gestion d'images : `tmvo_cleaning`. Cette opération est appelée chaque fois qu'il n'est plus possible d'allouer une entrée de la TMVO ou d'obtenir des segments ou blocs de mémoire partagée. L'opéartion balaie les entrées de la TMVO et, pour chaque objet non lié dans un domaine et sur lequel aucune transaction ne travaille, les actions suivantes sont réalisées :

- Si l'image été modifiée, elle est rangée en MPO. Appel à l'opération `mvo_storeObject` (cf. ci-après).
- L'objet est déverrouillé. Appel à la primitive `MPO_Release` (cf. section 6.3.6).
- Le segment ou bloc de segment de son image est libéré.
- L'entrée dans la TMVO est libérée.

On peut remarquer ici qu'entre le modèle de la récupération d'espace en MVO et son implantation, il y a une différence : Dans le modèle (cf. section 4.4), on ne récupère que l'espace des objets déverrouillés. Dans l'implantation, on va plus loin : On récupère l'espace des objets non liés dans un domaine. Ces objets sont, bien évidemment, déverrouillés avant la récupération d'espace.

La gestion d'images de chaque site dispose en outre d'une activité de sauvegarde pour ranger en MPO, de façon asynchrone, les images qui ont subi des modifications. Le code exécuté par cette activité balaie la TMVO et, pour chaque objet non lié dans un domaine et sur lequel aucune transaction ne travaille, son image est rangée en MPO (appel à `mvo_storeObject`).

La dernière opération interne du module de gestion d'images se charge de ranger en MPO l'image d'un objet : `mvo_storeObject`. Après avoir appelé la primitive `MPO_StoreObj` (cf. section 6.3.6), qui réalise effectivement le rangement, la variable qui permet de détecter une nouvelle modification est remise à son contenu initial[19].

6.3.4.3 Primitives

Les primitives du gestion des images sont les suivantes :

1. `TMVO_Init`
   Cette primitive permet de créer et initialiser le segment de mémoire partagée Unix utilisé pour la réalisation de la TMVO et de ses variables de gestion. Une option de la primitive permet simplement d'obtenir l'accès à ce segment.

---

19. En réalité, l'opération finit par la libération du segment ou bloc de segment de l'image. Ceci se rapporte à la manière dont les primitives `MPO_StoreObj` et `SO_StoreObj` sont réellement implantées (cf. sections 6.3.5.3 et 6.3.6).

2. `mvo_loadDescriptors`
   Cette primitive permet, d'une part, d'initialiser les descripteurs en MVO d'un objet et de sa classe à partir du contenu de leurs descripteurs en MPO et, de l'autre, d'obtenir en MVO des segments ou blocs de segment pour charger ultérieurement les états de l'objet et de sa classe.

3. `mvo_loadObject`
   Cette primitive permet de compléter la création d'une image, action commencée déjà par l'exécution de l'opération interne `loadDescr`.

6.3.5 Gestion d'objets stockés relative à un site

Ce module se charge de la gestion des blocs des objets stockés résidents dans un site. Les fonctions de gestion concernent aussi bien les blocs de descripteur que les blocs de données des objets stockés. L'allocation et la libération de ces blocs, ainsi que la gestion du cache sur le site, sont réalisées par les modules de support à la gestion des objets stockés.

L'arborescence des blocs d'un objet stocké est mise en oeuvre de la manière suivante :

Le descripteur en MPO d'un objet est contenu dans un bloc de descripteur. Si la taille de l'objet stocké correspondant est inférieure à la taille restant disponible dans ce bloc, l'état de l'objet est rangé dans le même bloc que son descripteur en MPO. Sinon, le reste du bloc de descripteur comporte les adresses des blocs de données contenant l'état.

Les deux derniers de ces blocs ne contiennent pas directement l'état de l'objet mais sont eux-mêmes des blocs d'indirection. L'indirection est simple pour l'avant dernier bloc de données. Elle est double (i.e., deux niveaux d'indirection) pour le dernier bloc.

Ceci correspond, pour une taille de bloc de 512 octets et une taille de descripteur en MPO de 256 octets, à une taille maximale d'objet stocké égale à 8287 KOctets (64*512 + 128*512 + 128*128*512). Ce mécanisme, similaire à celui utilisé pour les fichiers Unix, est illustré par la figure 6.7.

Les références-système comportent un indicateur de localisation dans la MPO d'un site (cf. section 4.2). Pour un objet, la valeur de cet indicateur comporte l'identificateur de son système d'objets de résidence et l'adresse, dans ce système d'objets, du bloc de descripteur qui maintient son descripteur en MPO. Cet accès direct aux objets stockés est dû au fait qu'on n'implante pas de table de descripteurs en MPO comme c'est le cas pour les descripteurs en MVO (cf. section 6.3.4.1).

Les opérations de la gestion d'objets stockés relative à un site sont les suivantes :

- Création d'un objet stocké. Cette opération consiste en la génération de l'oid d'un objet, en la création et initialisation de son descripteur en MPO et en l'allocation des blocs de données de son objet stocké.

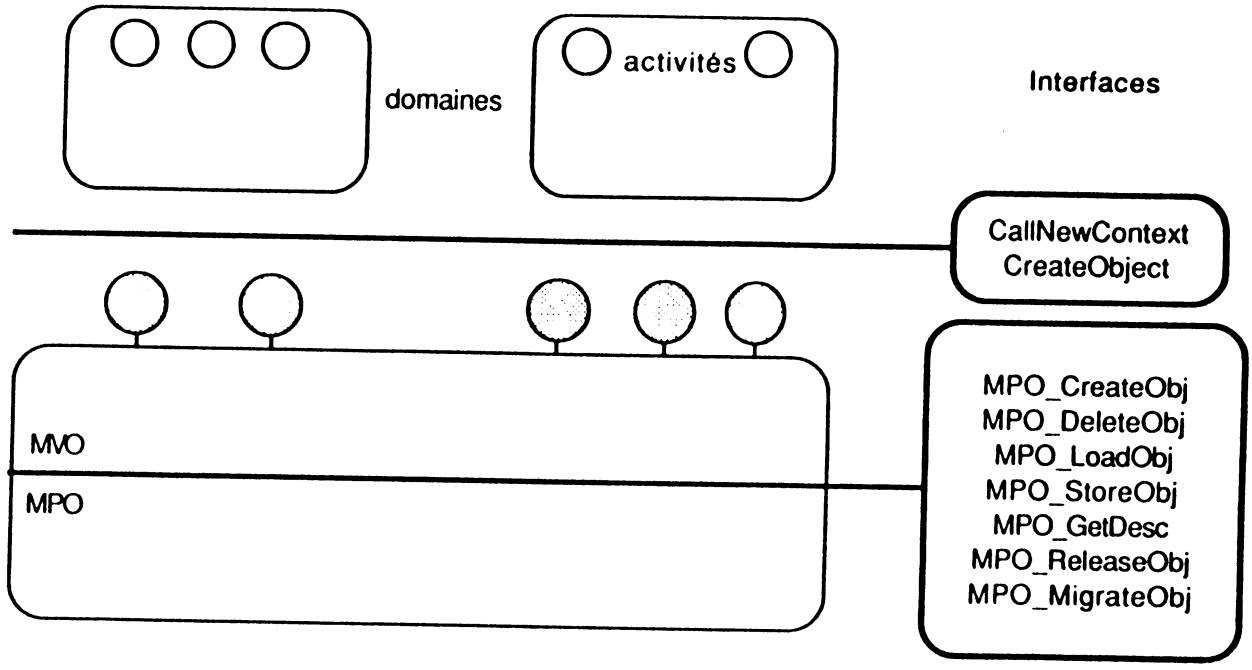


Figure 6.8 - Les deux interfaces de la machine à objets

1. `MPO_CreateObj`
   Création d'un objet stocké.
2. `MPO_DeleteObj`
   Destruction d'un objet stocké.
3. `MPO_LoadObj`
   Chargement en MVO d'un objet stocké.
4. `MPO_StoreObj`
   Rangement en MPO d'une image.
5. `MPO_GetDesc`
   Verrouillage d'un objet et validation des images d'un objet.
6. `MPO_ReleaseObj`
   Libération d'un objet.

Ces primitives réalisent la gestion répartie d'objets stockés. Elles se limitent donc à déterminer le site de résidence d'un objet stocké et, s'il s'agit d'un site distant, d'envoyer une requête à ce site pour contacter un processus-serveur. Le reste du travail est réalisé par les primitives de la gestion d'objets stockés relative à un site (cf. section 6.3.5).

La dernière primitive de l'interface que la MPO offre à la MVO est `MPO_MigrateObj`. Cette primitive permet à une activité de faire migrer, vers un

- Verrouillage d'un objet.
- Déverrouillage d'un objet.
- Validation des images d'un objet.

Un objet n'est pas verrouillé sur le site à la suite de la création de l'objet stocké correspondant. L'opération de destruction est, par contre, refusée lorsque l'objet est verrouillé sur le site.

6.3.5.1 Descripteur en MPO d'un objet

Un bloc de descripteur contient le descripteur en MPO d'un objet. Il s'agit des informations suivantes :

1. Son oid et une référence-système sur sa classe.
2. Les dates de création, de dernière modification et de dernier accès à l'objet stocké correspondant.
3. La taille de l'objet stocké.
4. Les variables utilisées pour le verrouiller, pour le déverrouiller et pour valider ses images. Ce sont les variables `NextSite`, `OwnerSite`, et `LastSite`.
5. Le contenu de l'objet stocké ou les adresses de ses blocs de données, selon que la taille de l'objet stocké est ou non inférieure à la taille restant disponible dans le bloc de descripteur.

6.3.5.2 Création et destruction d'objets stockés

La création d'un objet stocké se réalise en quatre étapes :

1. Allocation d'un bloc de descripteur pour le descripteur en MPO.
2. Initialisation de ce descripteur à partir des paramètres de l'opération de création.
3. Allocation des blocs de données nécessaires pour maintenir en MPO l'état de l'objet (blocs de l'objet stocké).
4. Initilisation de ces blocs de données.

Les trois premières étapes sont réalisées par la primitive `SO_CreateObj`. Une valeur négative est donnée à la taille de l'objet stocké lorsque les blocs de données sont alloués mais pas encore initialisés. Ceci permet de mettre en oeuvre l'optimisation de chargement montrée dans la transition d'état 8 de la figure 6.6.

La dernière étape est réalisée lors du premier rangement en MPO de l'image correspondante. Appel à la primitive `SO_StoreObj`.

Bien que les objets stockés garantisent la conservation des objets correspondants, la capacité limitée des ressources de stockage impose le besoin de récupérer l'espace que certains occupent en MPO. La destruction d'un objet stocké a lieu à la suite de l'appel à la primitive `SO_DeleteObj`. Cet appel est effectué seulement par les procédures qui mettent en oeuvre le ramasse-miettes (cf. section 4.4).

6.3.5.3 Primitives

Les primitives du module possèdent, exception faite de celle d'initialisation, les caractéristiques suivantes :

- Elles portent toutes sur un seul objet stocké.
- Elles sont exécutées par une activité ou par un processus-serveur sur le site de résidence de cet objet stocké.
- Leur premier paramètre est toujours un identificateur de site (cf. section 6.3.2) : Si la primitive est exécutée par une activité, il s'agit du site d'exécution de l'activité. Si la primitive est exécutée par un processus-serveur, ce site est le site où se déroule l'activité distante au profit de laquelle travaille le processus-serveur. Ceci permet de mettre en oeuvre les mécanismes de verrouillage, déverrouillage et validation qui s'appuient sur les identificateurs des sites.
- Elles sont appelées uniquement par les primitives correspondantes de la gestion répartie d'objets stockés (cf. sections 6.3.2 et 6.3.6).

La primitive d'initialisation s'exécute sur chaque site dont la mémoire secondaire est dédiée à l'implantation de systèmes d'objets.

Les primitives du module sont les suivantes :

1. `MPO_Init`
   Cette primitive permet d'initialiser les structures de gestion de la MPO d'un site. Les partitions Unix qui mettent en oeuvre les systèmes d'objets qui résident dans le site sont ouvertes et le cache du site est créé et initialisé. Une option de la primitive permet de formatter ces systèmes d'objets. Une autre option de la primitive permet simplement d'obtenir l'accès à ces structures de gestion.
2. `SO_CreateObj`
   Cette primitive permet la création d'un objet stocké. Elle rend une référence-système sur l'objet correspondant.
3. `SO_DeleteObj`
   Cette primitive permet de détruire un objet stocké. La destruction n'a pas lieu si l'objet correspondant possède une image.
4. `SO_LoadObj`
   Cette primitive permet de charger en MVO les blocs d'un l'objet stocké afin de créer une image de l'objet correspondant. Le chargement n'est possible que si

l'objet est verrouillé. Si la primitive est exécutée par un processus-serveur, les blocs sont envoyés à l'activité distante au profit de laquelle travaille ce processus-serveur.

5. `SO_StoreObj`
   Cette primitive permet de ranger en MPO, dans les blocs de données d'un objet stocké, le segment de l'image valide de l'objet correspondant. Une option de la primitive permet de réaliser le rangement de façon synchrone, c'est-à-dire, sans utiliser les fonctions du cache du site. La primitive ne retourne alors que lorsque l'image est effectivement écrite en mémoire secondaire. Une autre option permet de ranger l'image et de déverrouiller ensuite l'objet[20]. Si la primitive est exécutée par un processus-serveur, l'image est envoyée par l'activité distante au profit de laquelle travaille ce processus-serveur.

6. `SO_GetDesc`
   Cette primitive permet de verrouiller un objet sur un site. Une option de la primitive permet de le verrouiller tout en validant l'une de ses images. La primitive rend les informations de son descripteur en MPO et, s'il est déjà verrouillé, le site de verrouillage. L'algorithme de cette primitive est présenté dans l'annexe A.

7. `SO_ReleaseObj`
   Cette primitive permet de déverrouiller un objet. Celui-ci doit posséder une image pour pouvoir être déverrouillé.

6.3.6 Réalisation des interfaces de la machine à objets

Les interfaces de la machine à objets sont :

- Celle que la MPO offre à la MVO.
- Celle que la machine à objets offre à la gestion de domaines et d'activités.

Ces interfaces sont illustrées par la figure 6.8. Les fonctions de leurs primitives sont décrites par la suite.

L'interface que la MPO offre à la MVO

Les primitives de l'interface entre la MVO et la MPO sont les suivantes :

---

20. En réalité, la primitive à été implantée de la manière suivante : Le rangement de l'image, synchrone ou non, est toujours suivi du déverrouillage. L'option en question permet, en fait, de ranger l'image sans déverrouiller l'objet. Il a été choisi de faire ainsi puisque la probabilité de déverrouiller immédiatement après le rangement est normalement supérieure que celle du cas contraire.

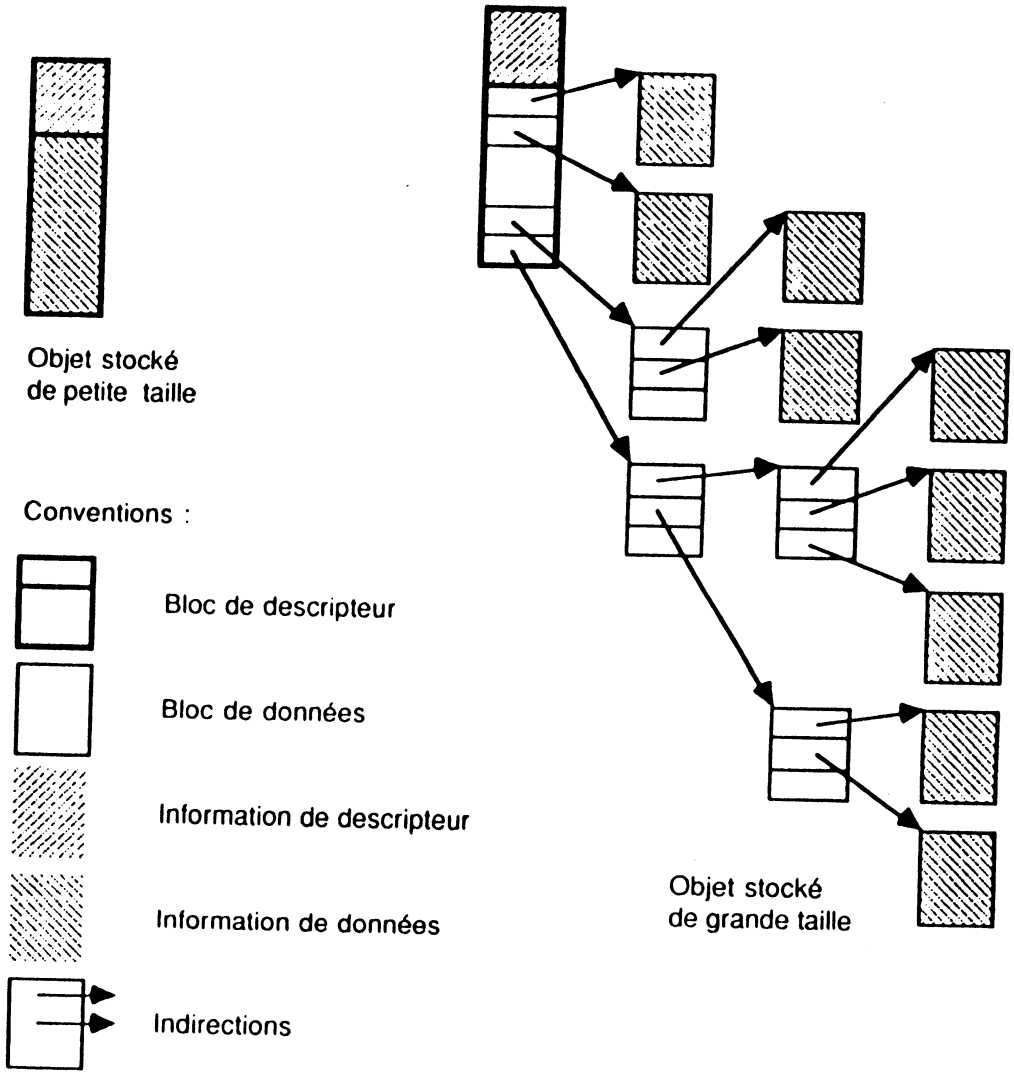


Figure 6.7 - Arborescence des blocs d'un objet stocké

- Destruction d'un objet stocké. Cette opération libère tous les blocs de l'objet stocké.
- Chargement en MVO d'un objet stocké. Le contenu de tous les blocs de données de l'objet stocké est chargé et regroupé dans un segement ou bloc de mémoire partagée alloué pour la création d'une image.
- Rangement en MPO d'une image. Le segment de l'image est rangé en répartissant son contenu entre les blocs de données de l'objet stocké correspondant.

Il a été décide que la gestion d'objets stockés relative à un site réalise aussi les fonctions suivantes :

nouveau système d'objets, l'objet stocké correspondant à un objet qui ne comporte pas d'image en MVO. Cette primitive n'a pas encore été implantée (cf. Conclusions).

L'interface de la machine à objets

Parmi les primitives de l'interface que la machine à objets offre à la gestion de domaines et d'activités, les seules qui se rapportent la mémoire d'objets sont les suivantes :

1. `CallNewContext`
   Interprétation, par un processeur virtuel, d'un appel de méthode sur un objet.
2. `CreateObject`
   Création, par un processeur virtuel, d'une instance d'une classe. L'objet créé est adressable par le processeur virtuel qui le crée.

Etant donné qu'elles relèvent d'un processeur virtuel et qu'un processeur virtuel correspond à l'ensemble des extensions d'une activité, ces deux primitives sont prises en compte par la gestion de domaines et d'activités. En conséquence, la primitive `CallNewContext` a été intégrée dans la primitive `guideCall` et la primitive `CreateObject` l'a été dans la primitive `guideCreate`.

6.3.7 Les processus qui exécutent la gestion de la mémoire d'objets

Sur un site, les processus Unix qui exécutent des primitives de la gestion de la mémoire d'objets sont les suivants :

- Le démon Guide.
- Les activités qui exécutent des méthodes d'objet. Le code des modules de la gestion de la mémoire d'objets est lié avec le code de chaque processus Unix réalisant une activité. Si le site où l'activité se déroule n'est le site de résidence d'aucun système d'objets, ni les deux modules de support à la gestion d'objets stockés ni le module de gestion d'objets stockés relative au site ne sont liés.
- L'activité de sauvegarde asynchrone d'images.
- Un processus de sauvegarde asynchrone de blocs de cache. Ce processus n'existe que si sur le site réside au moins un système d'objets.
- L'ensemble de processus-serveurs du site. Ces processus n'existent que si sur le site réside au moins un système d'objets.

Ces processus sont illustrés par la figure 6.9. Leurs fonctions concernant la gestion de la mémoire d'objets sont décrites par la suite.

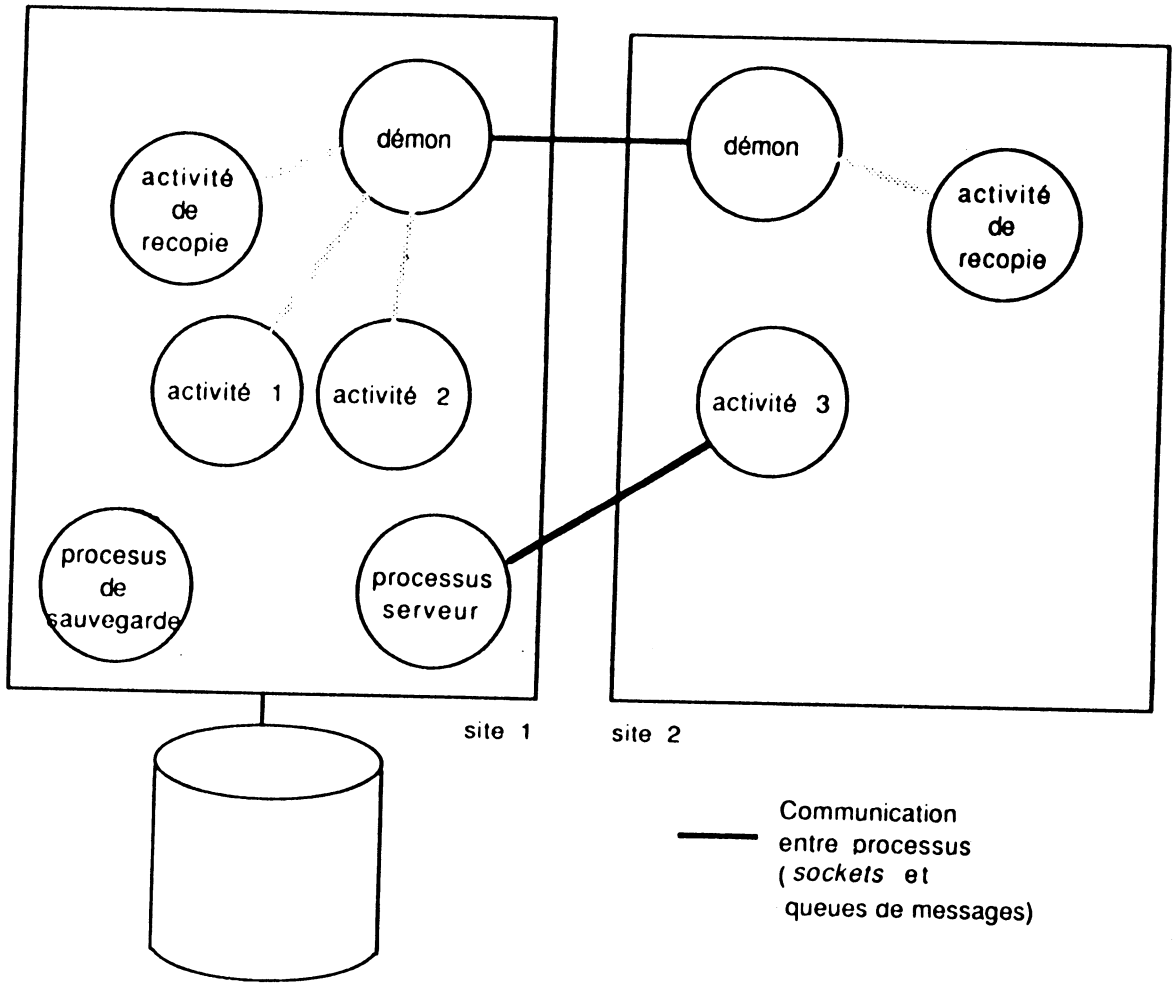


Figure 6.9 - Les processus de la gestion de la mémoire d'objets

Le démon

Les fonctions du démon du site, en ce qui concerne la gestion de la mémoire d'objets, sont les suivantes :

1. Lors de son démarrage :
   - Par l'exécution de la primitive `TMVO_Init`, il crée et initialise le segment de mémoire partagée correspondant à la TMVO du site. Si la gestion des segments en mémoire se réalise selon la version mono-segment, il crée et initialise le segment de mémoire partagée correspondant à la MVO du site. Il attache ces segments dans sa zone partagée.
   - Si sur le site réside au moins un système d'objets, il crée et initialise, par l'exécution de la primitive `MPO_Init`, le segment de mémoire partagée correspondant au cache du site. Il attache ce segment dans sa zone partagée.

Si nécessaire, les systèmes d'objets qui résident sur le site sont formattés.

- Il crée l'activité de sauvegarde asynchrone d'images.
- Si sur le site réside au moins un système d'objets, il crée le procesus de sauvegarde asynchrone de blocs de cache et les processus-serveurs du site.

2. Lors de son arrêt :
   - Il détruit toutes les activités du site.
   - Il détruit le processus de sauvegarde asynchrone de blocs de cache et les processus-serveurs du site.
   - Il détache tous les segments de mémoire partagée qu'il avait attachés lors de son démarrage.

Les activités

Les fonctions de gestion de la mémoire d'objets que toute activité réalise lors de son initialisation sont les suivantes :

1. Afin de pouvoir accéder aux entrées de la TMVO, elle exécute la primitive `TMVO_Init` avec l'option correspondante.
2. Si la gestion des segments en mémoire se réalise selon la version mono-segment, elle obtient l'accès à ce segment.
3. Si sur le site réside au moins un système d'objets, elle exécute, pour avoir accès au segment du cache du site, la primitive `MPO_Init` avec l'option correspondante.

Les options des primitives permettent à l'activité d'attacher, dans sa zone partagée, ces segments de mémoire partagée.

L'activité de sauvegarde d'images range en MPO, de façon asynchrone, les images des objet non liés et qui ont subi des modifications (cf. section 6.3.4.2).

Lors de son arrêt, toute activité détache de sa zone partagée tous les segments qu'elle avait attachés lors de son démarrage.

Les processus de sauvegarde et serveurs

Le processus de sauvegarde asynchrone écrit en mémoire secondaire, de façon asynchrone, les blocs modifiés du cache du site (cf. section 6.1.3.1).

Pour qu'ils puissent avoir accès au cache, le processus de sauvegarde asynchrone et les processus-serveurs exécutent, au moment de leur initialisation, la primitive `MPO_Init` avec l'option correspondante. Cette option leur permet d'attacher, dans leurs zones partagées, le segment de mémoire partagée qui implante le cache.

Ces processus détachent ces segments de mémoire partagée lors de leur arrêt.

6.4 L'interface de la machine virtuelle de Guide

Parmi les primitives de l'interface de la machine virtuelle, les seules qui se rapportent la mémoire d'objets sont les suivantes :

1. `guideCall`
   Interprétation d'un appel de méthode d'objet.
2. `guideCreate`
   Création d'une instance d'une classe.

Les fonctions de ces primitives sont présentées par la suite. Cependant, étant donné que plusieurs de ces fonctions ne concernent pas la mémoire d'objets, la description est faite d'une manière très schématique.

La primitive `guideCall`

L'exécution de cette primitive permet à une activité de réaliser l'appel à une des méthodes d'un objet. Ses fonctions sont les suivantes :

1. L'objet et sa classe sont liés dans le domaine de l'activité. Appel à la primitive `mvo_linkAsObject`. Si l'object est verrouillé sur un autre site, l'activité doit s'étendre sur ce site. L'extension exécute donc le `guideCall`.
2. Vérification du fait que la méthode est définie effectivement par la classe liée. Si ce n'est pas le cas [Guide-R2] [Krakowiak88], la super-classe de l'objet qui définit effectivement la méthode est recherchée et liée dans le domaine de l'activité. La classe de l'objet n'est pas déliée.
3. Construction des références-langage de l'objet et de la classe qui définit effectivement la méthode.
4. Détermination de l'adresse de branchement de la méthode à l'intérieur du code la classe qui la définit effectivement.
5. Sauvegarde du contexte d'exécution courant. Un contexte d'exécution comporte les références-langage et les descripteurs en MVO de l'objet sur lequel s'exécute une méthode et de sa classe.
6. Etablissement du nouveau contexte d'exécution.
7. Si la méthode est susceptible de modifier l'état de l'objet, préparation des mécanismes transactionnels pour le traitement des modifications.
8. Si la méthode comporte une condition de synchronisation qui n'est pas satisfaite, l'activité se bloque. Elle reste bloquée jusqu'à ce que la condition soit vraie [Decouchant88a].

9. Appel effectif de la méthode : branchement à l'adresse déterminée ci-dessus.
10. Déliaison du domaine de l'activité de l'objet, sa classe et la classe qui définit effectivement la méthode. Appels à la primitive `mvo_unLinkAsObject`.
11. Restitution de l'ancien contexte d'exécution.

La primitive `guideCreate`

L'exécution de cette primitive permet à une activité de créer une instance d'une classe. Le service d'administration des objets n'étant pas encore mis en place, les critères de choix du système d'objets de création (cf. section 4.1.2.1) ne peuvent pas être exprimés. En conséquence, le site de résidence de ce système d'objets doit être indiqué explicitement.

Les fonctions d'un `guideCreate` sont les suivantes :

1. La classe est liée dans le domaine de l'activité afin de pouvoir récupérer la taille de ses instances. Elle est ensuite déliée.
2. Le système d'objets de création de l'instance est déterminé à partir du nom du site et des informations de configuration du système.
3. L'objet stocké correspondant est créé en MPO. Appel à la primitive `MPO_CreateObj`.

A la sortie de cette primitive, ni l'objet ni sa classe ne sont maintenus liés dans le domaine de l'activité. L'objet sera lié au premier `guideCall` qui l'utilisera. C'est à ce moment que sa première image sera créée.

7. Réalisations des services

PP Les services de désignation symbolique et de gestion de versions ont été mis en oeuvre comme des applications du système. Ils ont été codés en langage Guide. Les types des objets OR et GV sont présentés dans les sections suivantes. Ils sont appelés, respectivement, `Directory` et `VersManager`. Les classes qui les réalisent sont décrites dans les annexes B et C.

Un des aspects où le langage et le système Guide facilitent le plus la programmation est la complète disparition des problèmes de distribution. Les services ont pu être programmés comme s'ils ne manipulaient que des objets locaux à un site, alors qu'ils concernent des objets qui peuvent être répartis sur tous les sites du réseau.

Les espaces des désignations sont globaux à tous les sites. Un objet peut toujours être désigné par la même expression de désignation quel que soit le site de création de son image ou le site de résidence de son objet stocké. La désignation symbolique dans Guide est analogue à celle utilisée dans le système Locus [Popek85].

Ce chapitre décrit uniquement la première version de la réalisation de chaque service. Des améliorations qui commencent à être entreprises seront introduites dans la deuxième version (cf. section Conclusions.1).

7.1 Description d'un objet répertoire

Le type d'un OR est présenté par la suite. Les méthodes qui réalisent la gestion des ORs eux-mêmes (y comprise la manipulation de leurs noms symboliques) ont été séparés de ceux qui se chargent de la gestion des noms symboliques des autres objets.

```
TYPE Directory IS

SYNONYM DesigExp = String ;
SYNONYM StringList = LIST OF String ;

METHOD Init (IN REF Directory) ;
METHOD Catalog (IN DesigExp ; REF Top) ;
METHOD Lookup (IN DesigExp) : REF Bottom ;
METHOD Remove (IN DesigExp) ;
METHOD Rename (IN DesigExp ; DesigExp) ;
METHOD List (IN DesigExp ; DesigExp) : REF StringList ;
METHOD MakeDir (IN DesigExp) : REF Directory ;
METHOD LookDir (IN DesigExp) : REF Directory ;
METHOD RemoveDir (IN DesigExp) ;
METHOD LinkDir (IN DesigExp ; DesigExp) : REF Directory ;
METHOD ListDir (IN DesigExp ; DesigExp) : REF StringList ;
```

```
METHOD Change (IN REF Top) ;
METHOD ChangeDir (IN REF Top) ;
METHOD Get :  REF Bottom ;
METHOD GetDir : REF Bottom ;

END Directory.
```

Les méthodes réalisent les fonctions suivantes :

1. `Init` :
   Initialisation des structures de données internes. La référence-système de l'OR qui crée cet OR et passée en paramètre (cf. section 5.1.2).
2. `Catalog` :
   Enregistrement, dans l'état de cet OR, de l'association entre l'expression de désignation et la référence-système passées en paramètre.
3. `Lookup` :
   Obtention de la référence-système de l'objet désigné par l'expression de désignation passée en paramètre.
4. `Remove` :
   Destruction de l'association existante entre l'expression de désignation passée en paramètre et une référence-système.
5. `Rename` :
   Changement du nom symbolique d'un objet. Le premier paramètre comporte le nom courant de l'objet et le deuxième comporte le nouveau nom. Les chemins d'accès compris dans ces deux expressions de désignation sont relatifs au même répertoire.
6. `List` :
   Obtention d'une liste de noms symboliques. Ces noms sont catalogués dans le répertoire dont l'OR correspondant est désigné par l'expression de désignation passée en premier paramètre. La deuxième expression de désignation comporte un atout qui permet de sélectionner les noms qui doivent apparaître dans la liste. La liste ne comporte pas de noms d'OR.
7. `MakeDir` :
   Création d'un répertoire. L'OR correspondant doit pouvoir être désigné par l'expression de désignation passée en paramètre. La méthode rend la référence-système de l'OR créé.
8. `LookDir` :
   Obtention de la référence-système de l'OR désigné par l'expression de désignation passée en paramètre.
9. `RemoveDir` :
   Destruction du répertoire dont l'OR correspondant est désigné par l'expression de désignation passée en paramètre. L'association existante entre cette expression de désignation et la référence-système de cet objet est détruite.

10. `LinkDir` :
    Création d'un lien au répertoire dont l'OR correspondant est désigné par l'expression de désignation passée en premier paramètre. Le répertoire qui met en oeuvre le lien doit être créé de façon à ce que son OR correspondant puisse être désigné par l'expression de désignation passée en deuxième paramètre. La méthode rend la référence-système de l'OR créé.

11. `ListDir` :
    Obtention d'une liste de noms symboliques d'OR. Ces noms sont catalogués dans le répertoire dont l'OR correspondant est désigné par l'expression de désignation passée en premier paramètre. La deuxième expression de désignation comporte un atout qui permet de sélectionner les noms qui doivent apparaître dans la liste. La liste ne comporte que de noms d'OR.

12. `Change, ChangeDir, Get, GetDir` :
    Méthodes de copie de l'état d'un OR . Elles sont utilisées pour la mise en oeuvre de la fonction de création d'un lien vers un répertoire (méthode `LinkDir`, cf. annexe B).

Dans la programmation de la classe qui réalise ce type, on a fait usage des objets internes de Guide. Les objets regroupés par les listes qui mettent en oeuvre un OR sont internes. L'état d'un OR *x* contient :

- Une référence-système sur l'OR qui l'a créé. Si *x* correspond à la racine de la hiérarchie de noms, son état comporte une référence sur lui-même.
- Une liste d'objets internes contenant chacun le nom symbolique et la référence-système d'un OR créé par les méthodes de *x*.
- Une liste d'objets internes contenant chacun le nom symbolique et la référence-système d'un objet non répertoire. Il s'agit des objets dont les noms symboliques sont catalogués dans le répertoire correspondant à *x*.

Cette structure relativement simple permet de construire un service offrant des fonctions de désignation symbolique analogues à celles du système de gestion de fichiers d'Unix. La séparation des associations <nom symbolique, référence-système> en deux listes résulte dans une interface manipulant explicitement soit les noms des objets, soit les noms des répertoires et les répertoires eux-mêmes.

Le code de la classe `Directory`, qui réalise le type `Directory`, est montré dans l'annexe B.

7.2 Description d'un gestionnaire de versions

Le type d'un GV est le suivant :

```
TYPE VersManager ;

SYNONYM DesigExp = String ;
```

```
METHOD Init ;
METHOD CreateVers (IN REF Top ; DesigExp) : REF Bottom ;
METHOD DeleteVers (IN REF Top ; DesigExp) : REF Bottom ;
METHOD GetVers (IN REF Top ; DesigExp) : REF Bottom ;

END VersManager.
```

Le paramètre des trois dernières méthodes correspond à la référence-système de l'objet qui les appelle. Ceci est nécessaire pour le traitement de l'expression de désignation `self`.

Les méthodes réalisent les fonctions suivantes :

1. `Init` :
   Initialisation des structures de données.
2. `CreateVers` :
   Création d'une nouvelle version à partir de la version désignée par l'expression de désignation passée en paramètre. La méthode rend la référence-système de la version créée.
3. `DeleteVers` :
   Destruction de la version désignée par l'expression de désignation passée paramètre. La méthode rend la référence-système de la version détruite.
4. `GetVers` :
   Obtention de la référence-système de la version désignée par l'expression de désignation passée en paramètre.

En opposition au service de désignation symbolique, les objets de la classe qui réalise le type `VersManager` comportent des listes d'objets persistants. L'état du GV d'un objet contient :

- Une référence-système vers la liste qui met en oeuvre l'historique des versions de l'objet. Un élément de l'historique comporte les informations suivantes concernant la version *v* :
  - — La référence-système de *v*.
  - — La référence-système du noeud correspondant à *v* dans l'arbre de dérivation.
- Une référence-système vers la racine de l'arbre de dérivation des versions de l'objet. Un noeud de l'arbre de dérivation contient les informations suivantes concernat la version *v* :
  - — La référence-système de *v*.
  - — La position dans l'historique de l'élément correspondant à *v*.
  - — La référence-système de la liste qui regroupe les noeuds de l'arbre de dérivation correspondants aux versions créées à partir de *v*.
  - — La référence-système du noeud de l'arbre de dérivation correspondant à la version mère de *v*.

L'historique et l'arbre de dérivation se référencent mutuellement parce que toute modification (addition ou suppression de versions) sur l'un entraîne forcément des modifications sur l'autre. Ces deux structures permettent de mettre en oeuvre la désignation de versions décrite antérieurement (cf. section 5.2.2). Une sous-classe de la classe prédéfinie `LIST` a du être implantée compte tenu que les listes standard de Guide ne pourvoient pas le parcours en arrière [Guide-R7] et que ceci est nécessaire pour l'implantation de l'expression de désignation `previous`.

Le code de la classe `VersManager`, qui réalise le type `VersManager`, est montré dans l'annexe C.

Conclusions

1. Les acquis du travail

Les objectifs de ce travail étaient la conception et la mise en oeuvre d'une mémoire d'objets pour un système d'exploitation réparti. Nous nous sommes aussi proposé de réaliser les fonctions de désignation symbolique d'objets et de gestion de versions des objets. Ces fonctions donnent lieu à deux services du système programmés en langage Guide.

La mémoire du système Guide comporte deux niveaux. Celui de la MVO supporte l'exécution des méthodes des objets. Le niveau de la MPO garantit leur conservation. L'unité d'échange entre les niveaux est la représentation de l'objet : l'image ou l'objet stocké. Le choix de cette unité est dû à plusieurs raisons : D'abord, Guide est un système à objets. On a considéré que, si l'interface entre la MVO et la MPO respecte l'objet comme un tout, il nous serait plus facile de mettre en évidence les problèmes liés à la gestion des objets. Ensuite, une unité d'échange comme la page demande l'adaptation d'une mémoire virtuelle paginée à la gestion d'objets. Or, une telle adaptation ne rentre pas dans nos objectifs. Enfin, puisque les objets Guide se veulent partagés, il aurait fallu établir une correspondance entre le partage des objets et le partage des unités d'échange choisies. Ceci ne fait pas partie non plus de nos buts.

Ainsi, lorsqu'une activité veut exécuter l'une des méthodes d'un objet, celui-ci et sa classe doivent se trouver entièrement en MVO. En plus, la demande de chargement d'un objet n'occasione que son chargement et celui de sa classe, si celle-ci n'est pas encore chargée. On ne fait pas de manipulation d'objets par blocs et on ne regroupe (pour l'instant) pas les objets pour les charger ensemble.

Le deuxième choix a été celui de verrouiller les objets sur un site afin d'interdire l'utilisation parallèle de plusieurs images. Les classes sont exemptes de cette 'restriction'. La raison de ce choix est la volonté de forcer les domaines à s'étendre pour mettre en oeuvre le mécanisme de partage par synchronisation sur un site. Ceci évite la réalisation d'une gestion de la cohérence des éventuelles copies créées sur des sites différents.

Lorsqu'un défaut d'objet est détecté, l'objet est verrouillé. Ensuite, l'objet stocké correspondant est chargé en MVO. L'image est alors consultée et éventuellement modifiée. Après la libération de l'objet, l'image peut rester en MVO tant que son espace n'est pas demandé par la gestion de la mémoire libre. Celle-ci la détruit après l'avoir rangée en MPO, s'il y avait des modifications. Tout chargement d'objet stocké a lieu depuis son site de résidence et c'est sur ce site que se fait le rangement de l'image correspondante. On a donc réalisé, par la coopération répartie entre les deux niveaux de mémoire, une véritable mémoire virtuelle d'objets.

Le ramasse-miettes de cette mémoire n'a pour l'instant pas été mis en oeuvre. Bien qu'on commence a en avoir besoin, on ne sait pas encore quelle est la meilleur option de réalisation. Etant donné le besoin d'une solution à très court terme, la première réalisation du ramasse-miettes ne sera vraisemblablement pas celle de la récupération en MPO.

L'objet choisi comme racine de persistance dependra de l'algorithme à mettre en oeuvre. Indépendamment son type, la cohérence de la structure de sa fermeture transitive sera maintenue. En effet, la gestion de la mémoire respecte les cinq règles de préservation de l'intégrité identifiées par Thatte (cf. Introduction). Les trois premières sont assurées du fait que Guide est un langage de haut niveau traduit en C. La cinquième condition est garantie parce que le système n'a pas à répondre aux requêtes de l'utilisateur. Pour celui-ci, l'espace de mémoire est une autre ressource que Guide, fidèle à sa philosophie d'intégration, a 'virtualisée'. La réalisation de la quatrième condition aura lieu lorsque le ramasse-miettes fonctionnera.

Les objets de la mémoire du système Guide persistent donc jusqu'au moment de leur ramassage. Ceci fait de cette mémoire une mise en oeuvre du modèle de Thatte.

Les aspects que la répartition influence sont traités en Guide : La cohérence est maintenue parce que les modifications sur une image ne se font que sur un site et sont ramenées ensuite sur le site de résidence de l'objet correspondant. En ce qui concerne la désignation, l'espace global des références permet aux objets d'un site de pointer sur des objets distants. Le mécanisme de localisation réalise deux fonctions : D'une part, il trouve les objet sur lesquels pointent des références devenues erronées à cause des migrations et, de l'autre, il met à jour ces références au fur et à mesure de leur utilisation. Quant aux cycles de miettes, les algorithmes de ramasse-miettes prévus les prennent en compte.

Nous n'avons pas été fidèles au modèle de la mémoire dans certains détails de sa réalisation. On peut en citer quelques-uns :

- Création d'un objet. On ne réserve pas d'espace en MPO lorsqu'un objet stocké est créé. Ceci est fait au moment du rangement de sa première image.
- Validation d'images. On ne fait pas de validation car on ne conserve pas les images en MVO. Il n'y a pour l'instant pas de moyen d'indiquer à la gestion de l'espace qu'une image doit être conservée.
- Références d'exécution. On n'a pas mis en oeuvre ces références telles qu'elles ont été définies dans le modèle. On utilise séparément les informations de la référence-système et celles de localisation en MVO.
- Rangement en MPO. On range une image seulement lorsque l'objet correspondant n'est pas lié alors qu'on pourrait et même devrait le faire lorsqu'il est lié ou simplement verrouillé. On associe également le rangement à la libération alors qu'il s'agit de fonctions indépendantes.
- Machine à objets. Son interface n'a pas été mise en oeuvre telle qu'elle est décrite dans le rapport sur le modèle d'exécution [Guide-R3].

Les services de désignation et de gestion de versions ont été réalisés comme des applications programmées en langage Guide. Ils ont constitué des excellents moyens de validation des mécanismes de ce langage. La programmation a été aisée à cause de la 'virtualisation' des ressources d'exécution et de conservation et aussi du fait que la distribution est cachée.

La gestion de noms symboliques permet de définir des noms qui ressemblent à ceux des fichiers d'Unix. Par contre, la désignation que met en place le service de versions a très peu de traits communs avec celle du service homologue d'Unix (SCCS [SPIX86]). La première désignation est plus générale. En revanche, la gestion elle-même n'est pas aussi riche que celle de SCCS. Sa simplicité permet cependant d'utiliser le service de versions comme point de départ pour la mise en oeuvre de gestions de versions plus complexes, comme celles des systèmes Encore et Etic.

Le service de désignation symbolique a dû subir certaines modifications à cause de l'incorporation des transactions. Il s'agit de modifications internes, l'interface externe reste la même. Désormais, les seules manipulations autorisées sur les *directories* sont la lecture et l'écriture. Le reste des opérations se fait par le biais d'un objet particulier, le NS (*name server*), dont les méthodes donnent lieu à des transactions.

Problèmes de la réalisation actuelle

Le choix de l'objet comme unité d'échange entre la MVO et la MPO présente deux inconvénients : Bien qu'on puisse détecter qu'un objet a été modifié, il n'est pas possible de savoir sur quelle page a eu lieu la modification. Il faudrait pour ceci intervenir au niveau de la mémoire virtuelle d'Unix, ce qui ne rentre pas dans nos objectifs. En conséquence, on ne peut pas faire que les objets partagent en MPO les blocs non modifiés. Des mécanismes de partage de blocs entre versions, comme celui mis en oeuvre dans le SGBD Exodus [Carey86], ne sont donc pas possibles à réaliser, bien que la structure arborescente de blocs existe pour les objets en MPO.

L'autre inconvenient, déjà signalé dans l'état de l'art, concerne la cohérence des rapports que les objets maintiennent entre eux par le biais des références. L'écriture d'un objet sur disque ne suffit pas pour garantir la cohérence de ces rapports : les points de contrôle sont nécessaires. Cependant, dans une mémoire à objets, les points de contrôle sont pratiquement impossibles à réaliser. Nous avons déjà dit que les concepteurs de Guide considèrent que la cohérence fournie par les transactions est suffisante. Ce qu'on pourrait faire de toute manière est de permettre le rangement en MPO de l'image d'un objet lorsque celui-ci est lié ou simplement verrouillé. La théorie du modèle de la mémoire permet ceci mais, dans le code de l'implantation, seules les images des objets non liés peuvent être écrites en MPO et la primitive d'écriture réalise par défaut la libération.

La réalisation actuelle du système ne prend pas en compte la cohérence des objets manipulés par les activités qui meurent subitement (tuées selon la condition du bloc `CO_BEGIN-CO_END`, par exemple). La prochaine version de Guide va permettre aux activités de mettre le contenu de ses objets dans un état cohérent avant de mourir. Une

telle expression de volonté *in articulo mortis* pourra être paramétrée par l'utilisateur via le mécanisme d'exceptions [Guide-R3].

La gestion de la mémoire d'Unix a posé des problèmes. En mémoire secondaire, il nous a fallu nous contenter du mode *raw* car le système d'entrées/sorties n'a pas été conçu pour supporter des applications requérant un grand niveau de fiabilité. On est limité par le fait que les processus Unix, en ce qui concerne la mémoire virtuelle, sont des cellules complètement séparées et indépendantes alors qu'on voudrait un espace global partagé par tous les processus. La gestion de la version multi-segments de la mémoire partagée présente un problème : un segment ne peut pas être attaché sans arrêt à la même adresse. L'expérience montre qu'après plusieurs succès, un nouvel essai d'attachement ne marche pas. Nous sommes donc contraints à abandonner cette version de manipulation de la mémoire partagée. On conserve la version mono-segment mais celle-ci présente le problème de débordement de segment qu'Unix ne peut pas détecter.

Fonctions non encore mises en oeuvre

Apart le ramasse-miettes, les fonctions non encore réalisées sont la migration et la duplication d'objets. Cette section présente quelques idées pour leur mise en oeuvre :

Migration

On rappelle que la fonction de migration permet de savoir si un objet stocké a migré ou non et, si oui, sur quel système d'objets. Deux mécanismes sont envisagés pour sa mise en oeuvre :

1. Conserver, dans le système d'objets d'origine, l'identificateur du nouveau système d'objets. Le bloc de descripteur utilisé pour l'objet stocké dans le système d'objets d'origine ne peut donc pas être réalloué.
2. Rechercher en diffusion l'objet stocké dans les différents systèmes d'objets de Guide. Ce mécanisme évite une indirection, mais nécessite un support de communication permettant la diffusion.

Duplication

En cas de panne du volume principal d'un système d'objets, un des volumes secondaires est choisi pour le remplacer. Le choix se fait en fonction de la charge des sites dont les systèmes d'objets locaux comportent des volumes secondaires encore opérationnels. Ceci permet au système d'objets de continuer à fonctionner normalement, le rangement des images modifiées étant toujours possible.

Même en dehors d'une panne, le nombre de volumes d'un système d'objets peut varier en cours d'utilisation. Un volume peut être ajouté dynamiquement à un système d'objets. Si celui-ci comporte déjà d'autres volumes, une phase de synchronisation est lancée afin d'amener le nouveau au niveau des autres. Un volume peut également être rétiré dynamiquement. S'il est le principal, un autre volume est automatiquement désigné pour le remplacer. La notion de volume principal est donc totalement transparente.

La sauvegarde d'un système d'objets (*back-up*) utilise aussi le mécanisme d'insertion et retrait dynamique des volumes. Lorsqu'on désire effectuer une sauvegarde, il suffit d'ajouter un volume vierge et d'effectuer la phase de synchronisation. Une fois cette phase terminée, le volume est rétiré et archivé. Il suffira de l'insérer en premier pour repartir de cette sauvegarde.

2. Perspectives

Certains aspects de gestion d'objets n'ont pas été considérés jusqu'à présent soit parce qu'ils ne rentrent pas dans le cadre des objectifs immédiats du système, soit parce qu'on n'avait pas l'expérience néccessaire pour les attaquer. Il s'agit de la gestion d'objets de taille variable, de la réalisation des types d'accès propres aux applications de bases de données, et de la gestion d'objets dans un environnement hétérogène. On commence à regarder la manière dont ces aspects pourraient être intégrés dans le modèle de la mémoire.

Un autre aspect intéressant, mais qui ne concerne pas directement le modèle de la mémoire, est celui de la possible utilisation de la MPO par des systèmes autres que Guide. Il s'agirait de voir comment la MPO pourrait tenir le rôle que WiSS joue pour O_2 ou qu'ObServer joue pour Encore et Garden.

On ne peut pas considérer que l'environnement de Guide sera toujours homogène. Le schéma qu'on montrera dans la section suivante se présente comme une sorte d'habillage de la machine virtuelle qui permet de l'adapter à l'architecture des différents sites mais sans toucher à rien de son contenu interne. En fait, aucun des trois modèles ne sera modifié.

La dynamicité de taille doit être considérée afin de permettre aux objets de grandir ou de diminuer alors que leus méthodes sont exécutées, c'est-à-dire, lorsqu'ils sont présents en MVO. Il serait donc plus approprié de parler d''images de taille variable'. Un objet stocké ne change pas dynamiquement de taille : Après avoir été chargé entièrement, son image commence à grandir est à diminuer. Celle-ci est ensuite rangée entièrement en MPO.

Les aspects 'bases de données' sont considérés pour deux raisons : le besoin de regrouper les objets stockés et de les charger ensemble afin d'améliorer la performance du système et le besoin de certaines applications d'accéder par le contenu aux objets. Le premier besoin a été ressenti dans Guide depuis bien longtemps [Scioville87], mais c'est seulement après l'analyse de la performance de la première version de la MPO qu'on a commencé à faire des choix (cf. sections suivantes). La raison pour laquelle on s'intéresse aux accès par le contenu est le fait qu'une partie des travaux du projet Guide se fait dans le cadre du projet Comandos, dont l'un des objectifs est de supporter les procédures de bureau. Or, dans celles-ci, les accès par le contenu sont nécessaires.

Hétérogénéité

Les travaux sur l'hétérogénéité dans Guide ont commencé par la définition d'un langage intermédiaire, nommé 'Smile'. Le langage Guide sera traduit dans Smile et celui-ci dans le code machine de chaque site du réseau hétérogène. Le format de Smile est basé sur le concept de *label*, point d'échappement qui permet la migration de l'exécution d'une activité.

Par opposition aux migrations qui se font dans le cadre d'un appel procédural à distance, le contexte d'exécution ne bouge pas de site lors du traitement d'un *label*. Une fois l'exécution lancée sur le site distant, un mécanisme de *trap* permet de demander et récupérer les variables dont on a besoin. Trois désignations sont nécessaires :

- Désignation des activités.
- Désignation des piles associées aux procédures (appels de méthode) des activités.
- Désignation des variables maintenues dans les piles.

L'espace des *labels* de Smile est plat. Ceci permet l'interruption et la migration de l'exécution à un moment quelconque de l'activité.

Images de taille variable

Une image de taille variable comporte une partie fixe, appelée 'descripteur', et une partie variable composée elle-même de deux blocs de mémoire. Le premier de ces blocs, appelé 'primaire', contient l'image telle qu'elle fut rangée en MPO la dernière fois. L'autre bloc met en oeuvre une table dont les entrées pointent sur des blocs de débordement. Lorsque l'image grandit, un nouveau bloc de débordement est alloué et une nouvelle entrée est ajoutée à la table.

Le fait d'associer à l'image un 'descripteur' de taille fixe permet la répresentation d'une image de taille variable par une image de taille fixe. Ceci autorise aussi la définition d'objets internes de taille variable.

Quand l'image est rangée en MPO, l'ensemble des blocs de débordement est 'concaténé' à la suite du primaire. La table et l'ensemble des blocs de débordement sont détruits.

Les détails du mécanisme proposé se trouvent dans [Rousset de Pina89]. Ce mécanisme est similaire à la gestion de la mémoire principale du serveur ObServer et ses clients (cf. section 2.1.2.3).

Aspects 'bases de données' dans Guide

On ne veut pas faire de Guide un SGBD à objets. On a donc choisi la voie d'ajouter au système des caractéristiques des SGBDs mais pas toutes. Un aspect laissé de côté est, par exemple, la gestion des contraintes d'intégrité.

Les modèles à objets encapsulent l'état des objets en y autorisant l'accès seulement par le biais des méthodes. En Guide, l'accès direct à certaines données internes, comme les compteurs de références et les références elles-mêmes, est autorisé. Il s'agit cependant d'un accès que le système lui-même réalise et qui est transparent à l'utilisateur.

L'accès par le contenu est voulu par l'utilisateur et donc forcément visible. Afin de l'autoriser, le modèle de données de Guide a été enrichi des **variables visibles**. Ce sont des variables qui font partie de l'état d'un objet mais qui sont définies dans son type et non dans sa classe. Des méthodes implicites de lecture et d'écriture permettent d'y accéder directement. Les variables définies dans la classe restent encapsulées par les méthodes.

Un nouveau constructeur de types, la **collection**, a été defini. Il permet de regrouper des objets d'un type afin d'accéder directement aux variables visibles définies dans ce type. Dans un premier temps, les collections sont contraintes à regrouper des objets de la même classe. En fait, une collection peut regrouper un sous-ensemble quelconque de l'ensemble des instances d'une classe. De cette manière, il est possible de définir le résultat d'un accès associatif sur une collection comme une nouvelle collection.

Les méthodes d'un objet de type collection permettent d'insérer un objet dans la collection, d'en supprimer un objet et de sélectionner le sous-ensemble dont les éléments vérifient un critère. Il est aussi possible de parcourir les éléments de la collection et de réaliser des opérations ensemblistes. On considère aussi la possibilité d'exprimer au niveau de la collection l'exécution de la même méthode sur tous les objets.

Les requêtes de manipulation des éléments d'une collection sont des objets de type **prédicat**. Il sont les paramètres des méthodes de la collection. Après interprétation, le prédicat est appliqué sur chaque élément pour déterminer le résultat. Des appels de méthode sur les objets collectionnés peuvent faire partie d'un prédicat. Les types collection et prédicat rejoignent l'idée de base de la réalisation des services de désignation symbolique et de gestion de versions : Un objet est utilisé pour regrouper d'autres objets et pour les désigner d'une manière particulière ; dans ce cas, la collection regroupe et la désignation est exprimée par le biais des prédicats.

Les collections seront rangées en MPO de la manière suivante : Une collection sera définie implicitement sur une classe afin de regrouper en MPO les objets stockés correspondants à *toutes* ses instances[21]. Les autres collections définies sur la classe seront des sous-ensembles de cette collection. Des index, uniques ou non, seront définis pour accéder aux objets stockés regroupés dans une collection implicite. Des méthodes de la collection sont prévues à cette fin.

---

21. Cette collection implicite met en oeuvre le concept de *cluster*.

Le regroupement en MPO devra être pris en compte dès que les objets stockés sont créés. Le mécanisme de stockage de collections provoquera certainement des changements sur la gestion de la mémoire. Deux voies sont possibles :

1. Modification de l'interface entre la MVO et la MPO pour prendre en compte l'accès aux objets d'une collection. Les objets stockés regroupés dans la collection implicite sont chargés en mémoire principale dans un cache particulier. Il sont passés ensuite à la MVO de manière traditionnelle. Par conséquent, il faut verrouiller et lier chaque objet de la collection. L'accès aux objets reste l'accès classique mais le nombre de `guideCalls` est important.

2. Les instances d'une classe collectionnable sont conservés en deux parties dans la MPO : La première, qui contient les variables visibles, est conservée dans l'état de la collection implicite. La deuxième partie, stockée normalement, contient le reste de l'état plus une référence à la collection implicite et un déplacement dans celle-ci. Dans le cas d'un accès individuel, la MPO reconstruit l'objet stocké. Les accès associatifs se font directement sur l'image de la collection implicite. Ceci diminue le nombre de `guideCalls` mais des nouveaux problèmes apparaissent : Les objets de la collection sont relégués à la catégorie d'internes et ne sont donc pas vus en tant qu'objets par la MVO. Des méthodes d'accès et des contrôles de concurrence particuliers doivent être conçus. La taille des collections implicites devient très grande.

Les détails des aspects 'bases de données' dans Guide se trouvent dans [Exertier89].

Utilisation de la MPO par un système différent de Guide

La MPO peut être utilisée comme le support de conservation de tout système qui respecte son interface. Cependant, dans l'état de sa réalisation actuelle, elle peut donner lieu à des conflits entre ces systèmes. Des risques d'incohérence menaceraient donc les objets stockés qu'elle conserve.

Malgré sa relative indépendance, c'est la MPO qui détermine si les objets peuvent être chargés ou non en MVO et c'est elle qui sait où ils sont chargés. Considérons qu'un système différent de Guide veut charger en mémoire principale un objet stocké de la MPO et que celle-ci autorise le chargement. Si une activité Guide veut ensuite accéder à l'objet, le domaine de l'activité doit diffuser sur le site *indiqué par la MPO*. Sur un même site donc, deux systèmes distincts veulent partager une image. Il faut que l'autre système ait des fonctions de partage, de synchronisation et de gestion de la mémoire principale similaires à celles de Guide pour que le partage puisse avoir lieu correctement.

Considérons que les deux systèmes parviennent à réaliser le partage. Si, malgré celà, ils ne se synchronisent pas correctement, il se peut que l'un libère l'objet alors que l'autre utilise encore son image. Puisque la MPO voit la libération, elle peut autoriser la création d'autre image et ne prendra jamais en compte celle conservée par le deuxième système.

Ce sont évidemment des problèmes externes à la MPO mais occasionés par les informations qu'elle manipule. Pour les éviter, on pourrait partitionner l'espace des objets stockés en sous-espaces disjoints et associer chaque sous-espace à un système différent.

Références

[Almes85]
G. Almes, A. Black, E. Lazowska, J. Noe : *The Eden System : A Technical Review* ; IEEE Transactions on Software Engineering, Vol. SE-11, No. 1, January 1985.

[Arctic88]
K. Birman, A. Herbert, S. Mullender, R. Needham, M. Satyanarayanan, M. Schroeder, A. Spector, W. Weihl : *Arctic'88 : an Advanced Course on Distributed Systems* ; Preliminary Lecture Notes, Participants Edition, University of Tromsø, Tromsø, Norway, July 1988.

[Atkinson83]
M. Atkinson, P. Bailey, K. Chisholm, P. Cockshott, R. Morrison : *An approach to persistent programming* ; The Computer Journal, Vol. 26, No. 4, December 1983.

[Atkinson85]
M. Atkinson : *PS-Algol Reference Manual* ; Persistent Programming Research Group, Department of Computer Sciences, University of Glasgow, Department of Computational Sciences, University of St. Andrews, 1985.

[Balter85]
R. Balter : *Maintien de la cohérence dans les systèmes d'information répartis* ; Thèse de Docteur d'Etat, USTMG, INPG, Grenoble, Juillet 1985.

[Balter86]
R. Balter, S. Krakowiak, M. Meysembourg, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandôme : *Principes de conception du système d'exploitation réparti Guide* ; Bigre+Globule, No. 52, Décembre 1986.

[Bancilhon88]
F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard, F. Vélez : *The Design and Implementation of O_2, an Object-Oriented Database System* ; Proceedings of the OODBS II Workshop, S. Verlag, Bad Münster, W. Germany, September 1988.

[Birman87]
K. Birman, T. Joseph : *Exploiting Virtual Synchrony in Distributed Systems* ; Proceedings of the 11th ACM SIGOPS Symposium on Operating Systems Principles, ACM, November 1987.

[Black85]
A. Black : *Supporting Distributed Applications : Experience with Eden* ; Proceedings of the 10th ACM SIGOPS Conference, ACM, December 1985.

[Black86a]
A. Black : *Distribution and abstract types in Emerald* ; Technical Report, No. 86-02-04, University of Washington, February 1986.

[Black86b]
A. Black : *Object structure in the Emerald system* ; Technical Report, No. 86-04-03, University of Washington, April 1986.

[Bourne78]
S. Bourne : *The UNIX Shell* ; The Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978.

[Brown85]
M. Brown, K. Kolling, E. Taft : *The Alpine File System* ; ACM Transactions on Computer Systems, Vol. 3, No. 4, November 1985.

[Bui Quang86]
N. Bui Quang : *Aspects Dynamiques et Gestion du Temps dans les Systèmes de Bases de Donées Généralisées* ; Thèse de Docteur de l'INPG, Spécialité Informatique, Grenoble, Novembre 1986.

[Butler86]
M. Butler : *An approach to persistent LISP objects* ; COMPCON 86 Proceedings, IEEE, San Francisco, March 1986.

[Carey86]
M. Carey, D. DeWitt, J. Richardson, E. Shekita : *Object and File Management in the EXODUS Extensible Database System* ; Twelfth International Conference on Very Large Data Bases, Kyoto, August 1986.

[Chou85]
H. Chou, D. DeWitt, R. Katz, A. Klug : *Design and Implementation of the Wisconsin Storage System* ; Software - Practice and Experience, Vol. 15, No. 10, October 1985.

[Cockshott84]
P. Cockshott, M. Atkinson, K. Chisholm, P. Bailey, R. Morrison : *Persistent object management system* ; Software - Practice and Experience, Vol. 14, 1984.

[Comandos87]
Comandos (ESPRIT Project 834) : *Object Oriented Architecture* ; Project

Report, No. D1-t2.1-870904, September 1987.

[Decouchant87]
D. Decouchant : *Partage et migration de l'information dans un système réparti à objets* ; Thèse de Docteur de l'USTMG (Grenoble 1), Spécialité Informatique, Grenoble, Juin 1987.

[Decouchant88a]
D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill, X. Rousset de Pina : *A synchronization mechanism for typed objects in a distributed system* ; Workshop on object-oriented concurrent programming, San Diego, September 1988.

[Decouchant88b]
D. Decouchant, A. Duda, A. Freyssinet, M. Riveill, X. Rousset de Pina, G. Vandôme : *Guide : an implementation of the Comandos object-oriented architecture on Unix* ; EUUG Autumn Conference Proceedings, Cascais, Portugal, October 1988.

[Delobel82]
C. Delobel, M. Adiba : *Bases de données et systèmes relationnels* ; Dunod, Informatique, Paris, 1982.

[Deutsch84]
L. Deutsch, A. Schiffman : *Efficient Implementation of the Smalltalk-80 System* ; ACM 11th Principles of Programming Languages Conference, ACM, Salt Lake City, 1984.

[Dijkstra78]
E. Dijkstra, L. Lamport, A. Martin, C. Scholten, E. Steffens : *On-the-Fly Garbage Collection : An exercise in cooperation* ; Comm. of the ACM, Vol. 21, No. 11, November 1978.

[Dijkstra83]
E. Dijkstra, W. Feijen, J. Van Gasteren : *Derivation of a termination detection algorithm for distributed computations* ; Inf. Proc. Letters, No. 16, 1983.

[Exertier87]
F. Exertier : *Systèmes d'exploitation pour les systèmes de gestion de bases de données* ; Rapport de DEA Informatique, INPG, USTMG (Grenoble 1), Grenoble, Juin 1987.

[Exertier89]
F. Exertier : *Aspects BD dans Guide : Concepts et mise en oeuvre* ; Note interne, No. 36, Projet Guide, Grenoble, Avril 1989.

[Fauvet88]
M. Fauvet : *Etic : Un SGBD pour la CAO dans un environnement partagé* ; Thèse de Docteur de l'Université Joseph Fourier (Grenoble 1), Grenoble, Septembre 1988.

[Fishman87]
D. Fishman : *IRIS : an Object-Oriented DBMS* ; ACM Transactions on Office Information Systems, Vol. 5, No. 1, January 1987.

[Freyssinet87]
A. Freyssinet : *Désignation dans un système réparti & Application au système Guide* ; Rapport DEA Informatique, INPG, Université Joseph Fourier (Grenoble 1), Grenoble, Juin 1987.

[Fridrich81]
M. Fridrich, W. Older : *The Felix File Server* ; Proceedings of the 8th ACM Symposium on Operating Systems Principles, ACM, Pacific Grove, December 1981.

[Gifford79]
D. Gifford : *Weighted voting for replicated data* ; Proceedings of the 7th ACM Symposium on Operating System Principles, ACM, Pacific Grove, December 1979.

[Goldberg76]
A. Goldberg, A. Kay : *Smalltalk-72 Instruction Manual* ; Xerox Palo Alto Research Center, Palo Alto, March 1976.

[Goldberg83]
A. Goldberg, D. Robson : *Smalltalk-80 : The language and its implementation* ; Addison-Wesley Publishing Company, 1983.

[Guide-R1]
R. Balter, S. Krakowiak, M. Meysembourg, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandôme : *Principes de conception du système d'exploitation réparti Guide* ; Rapport de recherche, No. 1, Laboratoire de Génie Informatique, Centre de Recherche Bull, Grenoble, Avril 1987.

[Guide-R2]
S. Krakowiak, M. Meysembourg, M. Riveill, C. Roisin : *Modèle d'objets et langage du système Guide* ; Rapport de recherche, No. 2, Laboratoire de Génie Informatique, Centre de Recherche Bull, Grenoble, Novembre 1987.

[Guide-R3]
R. Balter, J. Bernadat, D. Decouchant, S. Krakowiak, M. Riveill, X.

Rousset de Pina : *Modèle d'exécution du système Guide* ; Rapport de recherche, No. 3, Laboratoire de Génie Informatique, Centre de Recherche Bull, Grenoble, Décembre 1987.

[Guide-R4]
A. Freyssinet, R. Scioville, G. Vandôme : *Gestion des objets persistants dans le système Guide* ; Rapport de recherche, No. 4, Laboratoire de Génie Informatique, Centre de Recherche Bull, Grenoble, Décembre 1987.

[Guide-R5]
D. Decouchant, X. Rousset de Pina : *Principes de réalisation du noyau d'exécution de Guide sur Unix system V* ; Rapport de recherche, No. 5, Laboratoire de Génie Informatique, Centre de Recherche Bull, Grenoble, Juin 1988.

[Guide-R6]
A. Freyssinet, R. Scioville, G. Vandôme : *Réalisation de la mémoire permanente d'objets dans le système Guide* ; Rapport de recherche, No. 6, Laboratoire de Génie Informatique, Centre de Recherche Bull, Grenoble, Juin 1988.

[Guide-R7]
M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin : *Manuel du langage Guide* ; Rapport de recherche, No. 7, Laboratoire de Génie Informatique, Centre de Recherche Bull, Grenoble, A paraître 1989.

[Guide-R8]
R. Scioville : *Gestion de versions dans le système Guide* ; Rapport de recherche, No. 8, Laboratoire de Génie Informatique, Centre de Recherche Bull, Grenoble, A paraître 1989.

[Guide-R9]
A. Freyssinet, G. Vandôme : *Spécification et réalisation du service de désignation dans le système Guide* ; Rapport de recherche, No. 9, Laboratoire de Génie Informatique, Centre de Recherche Bull, Grenoble, A paraître 1989.

[Hornik87]
M. Hornik, S. Zdonik : *A Shared, Segmented Memory System for an Object-Oriented Database* ; ACM Transactions on Office Information Systems, Vol. 5, No. 1, January 1987.

[Ingalls78]
D. Ingalls : *The Smalltalk-76 Programming System Design and Implementation* ; ACM 5th Annual Symposium on Principles of Programming Languages, ACM, January 1978.

[Kaehler81]
T. Kaehler : *Virtual Memory for an Object-Oriented Language* ; BYTE, Vol. 6, No. 8, August 1981.

[Kaehler86]
T. Kaehler : *Virtual Memory on a Narrow Machine for an Object-Oriented Language* ; OOPSLA'86 Proceedings, ACM, Portland, September 1986.

[Kim88]
W. Kim, N. Ballou, H. Chou, J. Garza, D. Woelk, J. Banerjee : *Integrating an Object-Oriented Programming System with a Database System* ; OOPSLA'88 Proceedings, ACM, San Diego, September 1988.

[Krakowiak85]
S. Krakowiak : *Principes des systèmes d'exploitation des ordinateurs* ; Dunod, Informatique, Paris, 1985.

[Krakowiak88]
S. Krakowiak, M. Meysembourg, M. Riveill, C. Roisin : *Modèle d'objets et langage pour la programmation d'applications réparties* ; Actes des Journées Francophones sur l'Informatique, Genève, Janvier 1988.

[Laforgue88]
P. Laforgue, E. Paire : *Le réseau hétérogène multi-médias de l'IMAG* ; Proceedings de l'AFUU, 1988.

[Ledot88]
P. Ledot : *Un modèle transactionnel pour un système à objets* ; Rapport DEA Informatique, INPG, Université Joseph Fourier (Grenoble 1), Grenoble, Juin 1988.

[Lindsay84]
B. Lindsay, L. Haas, C. Mohan, P. Wilms, R. Yost : *Computation and communication in R* : A distributed database manager* ; ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984.

[Liskov83]
B. Liskov : *Guardians and Actions : Linguistic Support for Robust, Distributed Programs* ; ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, July 1983.

[Liskov88]
B. Liskov : *Distributed Programming in Argus* ; Comm. of the ACM, Vol. 31, No. 3, March 1988.

[Lunati88]
J. Lunati : *Migration de processus et partage de charge dans les systèmes répartis - A pplication au système Guide* ; Rapport DEA Informatique, INPG, Université Joseph Fourier (Grenoble 1), Septembre 1988.

[Lyon84]
B. Lyon, G. Sager, J. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, D. Walsh, P. Weiss : *Overview of the Sun Network File System* ; Sun Microsystems, Inc, October 1984.

[Lécluse89]
C. Lécluse, P. Richard : *The O_2 Database Programming Language* ; Technical Report Altaîr, No. 26-89, Altaîr, Le Chesnay, January 1989.

[Maier87]
D. Maier, J. Stein : *Development and Implementation of an Object-Oriented DBMS* ; Research Directions in Object-Oriented Programming, MIT Press, B. Shriver, P. Wegner, Cambridge, Massachussets, 1987.

[Meysembourg88]
M. Meysembourg, M. Riveill, C. Roisin : *Le modèle d'objets de Guide et sa mise en oeuvre répartie* ; Actes du colloque C3, Angoulême, Décembre 1988.

[Meysembourg89]
M. Meysembourg : *Modèle et langage à objets pour la programmation d'applications réparties* ; Thèse de Docteur de l'INPG, Spécialité Informatique, Grenoble, A paraître 1989.

[Mitchell82]
J. Mitchell, J. Dion : *A Comparison of Two Network-Based File Systems* ; Comm. of the ACM, Vol. 25, No. 4, April 1982.

[Morris86]
J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, F. Smith : *Andrew : A Distributed Personal Computing Environment* ; Comm. of the ACM, Vol. 29, No. 3, March 1986.

[Nelson83]
D. Nelson : *Informatique distribuée dans le système Domain d'Apollo* ; Convention Informatique, 1983.

[Nguyen Van89]
H. Nguyen Van : *Problèmes de Ramasse-miettes dans Guide* ; Note interne, No. 39, Projet Guide, Grenoble, Mars 1989.

[Nygaard86]
K. Nygaard : *Basic Concepts in Object-Oriented Programming* ; ACM SIGPLAN Notices Notices, Vol. 21, No. 10, October 1986.

[Oracle84]
Oracle Corporation : *Host Language Call Interface Manual* ; June 1984.

[Oracle85]
Oracle Corporation : *Database Administrator's Guide* ; April 1985.

[Organick72]
E. Organick : *The Multics system : an examination of its structure* ; MIT Press, 1972.

[Peterson77]
J. Peterson, T. Norman : *Buddy Systems* ; Comm. of the ACM, Vol. 20, No. 6, June 1977.

[Popek85]
G. Popek, B. Walker : *The LOCUS Distributed System Architecture* ; MIT Press, 1985.

[Pucheral88]
P. Pucheral, J. Thevenin, H. Steffen : *Geode : Un Gestionnaire d'Objets Extensible Orienté Grande Mémoire* ; Conférence du PRC-BD, Grenoble, Janvier 1988.

[Quint86]
V. Quint, I. Vatton, H. Bedor : *Le système Grif* ; TSI, Vol. 5, No. 4, 1986.

[Reiss86]
S. Reiss : *An object-oriented framework for graphical programming* ; ACM SIGPLAN Notices Notices, Vol. 21, No. 10, October 1986.

[Rentsch82]
T. Rentsch : *Object-Oriented Programming* ; ACM SIGPLAN Notices Notices, Vol. 17, No. 9, September 1982.

[Rieu86]
D. Rieu : *Nature, état et dynamicité de l'objet CAO* ; Actes des Deuxièmes Journées 'Bases de Données Avancées', INRIA, Giens, Var, Avril 1986.

[Rousset de Pina89]
X. Rousset de Pina : *Tableaux de taille variable en Guide* ; Note interne, No. 33, Projet Guide, Grenoble, Janvier 1989.

[SPIX86]
SPIX System Support Tools Guide : *Source Code Control System User Guide* ; Bull S.A., September 1986.

[Sandberg86]
R. Sandberg : *The Sun Network File System : Design, Implementation and Experience* ; EUUG Spring Conference Proceedings, Florence, Italy, April 1986.

[Satyanarayanan84]
M. Satyanarayanan : *The ITC Project : An Experiment in Large-Scale Distributed Personel Computing* ; Information Technology Center, Pittsburgh, PA-ITC-035, October 1984.

[Satyanarayanan85]
M. Satyanarayanan, J. Howard, A. Spector, M. West : *The ITC Distributed File System : Principles and Design* ; Information Technology Center, Pittsburgh, PA-ITC-039, May 1985.

[Scioville86]
R. Scioville : *Systèmes de Gestion de Fichiers Distribués (Résumé)* ; Note interne, No. 19, Projet Guide, Grenoble, Septembre 1986.

[Scioville87]
R. Scioville : *Relations entre objets* ; Note de travail, No. SGO.1, Projet Guide, Grenoble, Fevrier 1987.

[Shapiro85]
M. Shapiro, S. Habert, R. Dumeur, J. Fekete : *SOMIW Operating System Design* ; Technical Report, Esprit Project 367, December 1985.

[Skarra86a]
A. Skarra, S. Zdonik, S. Reiss : *An Object Server for an Object-Oriented Database System* ; OOPSLA'86 Proceedings, ACM, Portland, September 1986.

[Skarra86b]
A. Skarra, S. Zdonik : *The management of changing types in an object-oriented database* ; OOPSLA'86 Proceedings, ACM, Portland, September 1986.

[Spector88]
A. Spector, R. Pausch, G. Bruell : *Camelot : A Flexible Distributed Transaction Processing System* ; COMPCON 88 Proceedings, San Francisco, February 1988.

[Stamos84]
J. Stamos : *Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory* ; ACM Transactions on Computer Systems, Vol. 2, No.

2, May 1984.

[Sturgis80]
H. Sturgis, J. Mitchell, J. Israel : *Issues in the Design and Use of a Distributed File System* ; ACM SIGOPS Operating Systems Review, Vol. 14, No. 3, July 1980.

[Svobodova84]
L. Svobodova : *File Servers for Network-Based Distributed Systems* ; ACM Computing Surveys, Vol. 16, No. 4, December 1984.

[Thatte86]
S. Thatte : *Persistent Memory : A Storage Architecture for Object-Oriented Database Systems* ; Proceedings of the International Workshop on Object-Oriented Database Systems, IEEE, Pacific Grove, September 1986.

[Ungar84]
D. Ungar : *Generation Scavenging : A Non-disruptive High Performance Storage Reclamation Algorithm* ; First Symposium on Practical Software Development Environments, ACM, Pittsburgh, April 1984.

[Vélez89]
F. Vélez, G. Bernard, V. Darnis : *The O_2 Object Manager : an Overview* ; Technical Report Altaîr, No. 27-89, Altaîr, Le Chesnay, February 1989.

[West85]
M. West, D. Nichols, J. Howard, M. Satyanarayanan, R. Sidebotham : *The ITC Distributed File System : Prototype and Experience* ; Information Technology Center, Pittsburgh, PA-ITC-040, March 1985.

[Wupit83]
A. Wupit : *Comparison of Unix Network Systems* ; Université Catholique de Louvain, 1983.

[Zdonik86]
S. Zdonik, P. Wegner : *Language and methodology for object-oriented database environments* ; Proceedings of the 19th Annual Hawaii International Conference on System Sciences, Honolulu, January 1986.

Annexes :

A. Algorithme de verrouillage et validation

B. Réalisation du service de désignation symbolique

C. Réalisation du service de versions

Annexe A : Algorithme de verrouillage et validation

L'algorithme de la primitive `SO_GetDesc` **est le suivant :**

```
Primitive SO_GetDesc

/* Paramètre :
 * Référence système sur l'objet;
 * à partir de cette référence-système, on obtient
 * le descripteur de l'objet
 *
 * Paramètres par référence :
 * Site, pour rendre l'identificateur du site
 * de l'image active de l'objet
 * Size, pour rendre la taille de l'objet
 * Class, pour rendre la référence-système
 * de la classe de l'objet
 */

/* déverrouillage après panne
 * si le dernier accès est antérieur au boot
 */
if (Access < TimeBoot) {
 |   /* l'objet est déverrouillé */
 |   NextSite = OwnerSite = NOSITE
}
/* on ne peut pas verrouiller si
 * l'objet est chargé ou en cours de chargement sur mon site
 */
if (NextSite == MYSITE || OwnerSite == MYSITE) {
 |   return INVALID
}
/* on ne peut pas verrouiller si
 * l'objet est en cours de chargement sur un autre site
 * (i.e., il est déjà verrouillé) et si
 * le temps écoulé à partir du verrouillage n'est pas trop grand
 */
if (NextSite != NOSITE && TimeOut - CurrentTime < MPO_TIMEOUT) {
 |   return BUSY
}
/* on ne peut pas verrouiller si
 * l'objet est chargé sur un autre site
 */
if (OwnerSite != NOSITE) {
 |   Site = OwnerSite
 |   return SITE
}
/* on peut verrouiller */
if (MySize < 0) {
```

```
  |   /* objet créé mais non initialisé */
  |   Access = CurrentTime
  |   /* l'objet est considéré comme chargé */
  |   OwnerSite = MYSITE
  |   TimeOut = -1
  |   Size = MySize
  |   Class = MyClass
  |   return EMPTY
}
/* on peut verrouiller */
if (l'indicateur FRELOAD n'est pas positionné) {
  |   /* on ne demande pas de validation */
  |   Access = TimeOut = CurrentTime
  |   NextSite = MYSITE
  |   Size = MySize
  |   Class = MyClass
  |   return OK
} else {
  |   /* on demande une validation /*
  |   if (LastSite != MYSITE) {
  |   |   /* validation réfusée */
  |   |   Access = TimeOut = CurrentTime
  |   |   NextSite = MYSITE
  |   |   Size = MySize
  |   |   Class = MyClass
  |   |   return MODIFIED
  |   } else {
  |   |   /* validation acceptée */
  |   |   Access = CurrentTime
  |   |   /* l'objet est considéré comme chargé */
  |   |   OwnerSite = MYSITE
  |   |   TimeOut = -1
  |   |   Size = MySize
  |   |   Class = MyClass
  |   |   return OK
  |   }
}
```

Annexe B : Réalisation du service de désignation symbolique

La classe qui réalise le type `Directory` est la suivante :

```
CLASS Directory IMPLEMENTS Directory ;

SYNONYM Dir = RECORD [name, dirref] OF [String, REF Directory] ;
SYNONYM DirList = LIST OF Dir ;
SYNONYM Obj = RECORD [name, objref] OF [String, REF Top] ;
SYNONYM ObjList = LIST OF Obj ;
SYNONYM StringList = LIST OF String ;
SYNONYM DesigExp = String ;

father : REF Directory ;
// Objet répertoire qui a créé cet objet répertoire.
dir_list : REF DirList  ; obj_list : REF ObjList  ;
output : REF ChanOut ;
// Canal de sortie. Il s'agit d'un objet de classe prédéfinie.

PROCEDURE SearchDir (IN dir_name : String ; OUT dir : REF Dir) : Boolean ;
// Procédure qui permet de déterminer si un nom d'objet répertoire a été
// catalogué dans le répertoire correspondant à cet objet répertoire
BEGIN
dir_list.First ;
dir := dir_list.Next ;
WHILE dir # NIL DO
      IF dir_name = dir.name
         THEN RETURN TRUE ;
         ELSE dir := dir_list.Next ;
      END ;
END ;
RETURN FALSE ;
END SearchDir ;

PROCEDURE SearchObj (IN obj_name : String ; OUT obj : REF Obj) : Boolean ;
// Procédure qui permet de déterminer si un nom d'objet a été catalogué
// dans le répertoire correspondant à cet objet répertoire. Le code est le
// même que celui de la procédure antérieure sauf que la liste utilisée est
// obj_list
BEGIN
END SearchObj ;

METHOD Init (IN dir : REF Directory) ;
BEGIN
father := dir ;
END Init ;

METHOD Change (IN new_obj_list : REF Top) ;
// Changement de la valeur de obj_list
```

```
BEGIN
obj_list := new_obj_list ;
END Change ;

METHOD ChangeDir (IN new_dir_list : REF Top) ;
// Changement de la valeur de dir_list
BEGIN
dir_list := new_dir_list ;
END ChangeDir ;

METHOD Get :  REF Bottom ;
BEGIN
RETURN obj_list ;
END Get ;

METHOD GetDir : REF Bottom ;
BEGIN
RETURN dir_list ;
END GetDir ;

METHOD Catalog (IN sname : DesigExp ; object : REF Top) ;
dir : REF Dir ; tmp_obj , new_obj : REF Obj ;
dir_name, tmp_path : String ;
BEGIN
IF ((sname = NIL) OR (sname = ".") OR (sname = ".."))
   THEN output.WriteString("name_invalid0) ;
        RETURN ;
END ;
dir_name := sname.Scan("/") ;
// Si l'expression de désignation ne comporte pas le caractère '/', le nom est
// catalogué dans le répertoire correspondant à cet objet répertoire
IF dir_name = NIL
   THEN IF SearchDir(sname, dir)
           THEN output.WriteString("name_already_used0) ;
                RETURN ;
         ELSE obj_list.First ;
            tmp_obj := obj_list.Next ;
            WHILE  tmp_obj # NIL DO
                   IF sname = tmp_obj.name
                          THEN output.WriteString("name_already_used0) ;
                          RETURN ;
                   END ;
                   IF sname < tmp_obj.name
                          THEN new_obj := Obj.New ;
                          new_obj.name := sname ;
                          new_obj.objref := object ;
                          obj_list.Insert(new_obj) ;
                          RETURN ;
                   END ;
                   tmp_obj := obj_list.Next ;
            END ;
            new_obj := Obj.New ;
            new_obj.name := sname ;
```

```
            new_obj.objref := object ;
            obj_list.Insert(new_obj) ;
            RETURN ;
      END ;
   ELSE // Des appels récursifs
        tmp_sname := sname.Sub(dir_name) ;
      dir_name := dir_name.Rsub("/") ;
      IF dir_name = "."
           THEN SELF.Catalog(tmp_sname, object) ;
              RETURN ;
      END ;
      IF dir_name = ".."
           THEN father.Catalog(tmp_sname, object) ;
              RETURN ;
      END ;
      IF SearchDir(dir_name, dir)
           THEN dir.dirref.Catalog(tmp_sname) ;
              RETURN ;
         ELSE output.WriteString("path_not_found0) ;
              RETURN ;
      END ;
END ;
END Catalog ;

METHOD LinkDir (IN path : DesigExp ; new_path : DesigExp) : REF Directory ;
dir : REF Dir ; obj : REF Obj ;
new_dir, old_dir : REF Directory ;
new_dir_name, tmp_path : String ;
BEGIN
IF (( new_path = "." ) OR (new_path = NIL ) OR (new_path = ".." ))
   THEN output.WriteString("name_invalid0)
        RETURN NIL ;
END ;
new_dir_name := new_path.Scan("/") ;
IF new_dir_name = NIL
   THEN IF SearchObj(new_path, obj)
           THEN output.WriteString("path_already_used0) ;
            RETURN NIL ;
      END ;
      IF SearchDir(new_path, dir)
           THEN output.WriteString("path_already_used0) ;
            RETURN NIL ;
      END ;
      new_dir := SELF.MakeDir(new_path) ;
      old_dir := SELF.LookDir(path) ;
      new_dir.ChangeDir(old_dir.GetDir) ;
      new_dir.Change(old_dir.Get) ;
        // Ainsi établit-on le lien entre new_dir et old_dir
      RETURN new_dir ;
   ELSE //Des appels récursifs
      tmp_path := new_path.Sub(new_dir_name) ;
      new_dir_name := new_dir_name.Rsub("/") ;
      IF new_dir_name = "."
```

```
            THEN RETURN SELF.LinkDir(path, tmp_path) ;
        END ;
        IF new_dir_name = ".."
            THEN RETURN father.LinkDir(path, tmp_path) ;
        END ;
        IF SearchDir(new_dir_name, dir)
            THEN RETURN dir.dirref.LinkDir(path, tmp_path) ;
          ELSE output.WriteString("path_not_found0) ;
                RETURN NIL ;
        END ;
END ;
END LinkDir ;

METHOD LookUp (IN sname : DesigExp) : REF Bottom ;
METHOD LookDir (IN path : DesigExp) : REF Directory ;
METHOD Remove (IN sname : DesigExp) ;
METHOD Rename (IN sname : DesigExp ; new_sname : DesigExp) ;
METHOD List (IN path : DesigExp ; wildcard : DesigExp) : REF StringList ;
METHOD MakeDir (IN path : DesigExp) : REF Directory ;
METHOD RemoveDir (IN path : DesigExp) ;
METHOD ListDir (IN path : DesigExp ; wildcard : DesigExp) : REF StringList ;
// Le code de ces méthodes est présenté dans [Guide-R9]

END Directory.
```

Annexe C : Réalisation du service de gestion de versions des objets

Avant le code de la classe `VersManager`, on présente les types et les classes que celle-ci utilise. En fin d'annexe est aussi montré le code des méthodes ajoutées à la classe `Top` pour la gestion des versions.

Le sous-type et la sous-classe du type et de la classe `LIST` implantés pour introduire le parcours en arrière d'une liste sont les suivants :

```
TYPE ListWithPrev IS LIST
METHOD Previous : REF Bottom ;
END ListWithPrev.

CLASS ListWithPrev IS LIST
IMPLEMENTS ListWithPrev ;
METHOD Previous : REF Bottom ;
// Implantation de la fonction qui permet de faire marche arrière dans
// une liste. Son code est présenté dans [Guide-R8]
END ListWithPrev.
```

Le type et la classe d'un élément de l'historique des versions d'un objet sont les suivants :

```
TYPE HistItem IS
METHOD GetObject : REF Bottom ;
METHOD GetDerivItem : REF DerivItem ;
METHOD PutVersion (IN REF Top ; REF DerivItem) : Integer ;
END HistItem.

CLASS HistItem IMPLEMENTS HistItem ;
theObject : REF Top ; theDerivItem : REF DerivItem ;

METHOD GetObject : REF Bottom ;
BEGIN
RETURN theObject ;
END GetObject ;

METHOD GetDerivItem : REF DerivItem ;
BEGIN
RETURN theDerivItem ;
END GetDerivItem ;
```

```
METHOD PutVersion (IN anObject : REF Top ;
                   aDerivItem : REF DerivItem) : Integer ;
BEGIN
IF ((anObject = NIL) OR (aDerivItem  = NIL))
   THEN RETURN -1 ;
   ELSE theObject := anObject ;
        theDerivItem := aDerivItem ;
        RETURN 0 ;
END ;
END PutVersion ;

END HistItem.
```

Le type et la classe d'un noeud de l'arbre de dérivation des versions d'un objet sont les suivants :

```
TYPE DerivItem IS
METHOD GetObject : REF Bottom ;
METHOD GetPosition : Integer ;
METHOD GetDaughters : REF ListDerivItem ;
METHOD GetMother : REF DerivItem ;
METHOD PutVersion (IN REF Top ; Integer ;
            REF ListDerivItem ; REF DerivItem) : Integer ;
END DerivItem.

CLASS DerivItem IMPLEMENTS DerivItem ;
theObject : REF Top ; thePosition : Integer ;
theDaughters : REF ListDerivItem ; theMother : REF DerivItem ;

METHOD GetObject : REF Bottom ;
BEGIN
RETURN theObject ;
END GetObject ;

METHOD GetPosition : Integer ;
BEGIN
RETURN thePosition ;
END GetPosition ;

METHOD GetDaughters : REF ListDerivItem ;
BEGIN
RETURN theDaughters ;
END GetDaughters ;

METHOD GetMother : REF DerivItem ;
BEGIN
RETURN theMother ;
END GetMother ;
```

```
METHOD PutVersion (IN  anObject : REF Top ; aPosition : Integer ;
        someDaughters : REF ListDerivItem ; aMother : REF DerivItem) : Integer ;
BEGIN
IF ((anObject = NIL) OR (aPosition < 0))
   THEN RETURN -1 ;
   ELSE theObject := anObject ;
        thePosition := aPosition ;
        theDaughters := someDaughters ;
        theMother := aMother ;
        RETURN 0 ;
END ;
END PutVersion ;

END DerivItem.
```

Le type et la classe de l'historique des versions d'un objet sont les suivants :

```
TYPE Historic IS ListWithPrev
METHOD First : REF HistItem ;
METHOD Next : REF HistItem ;
METHOD Last : REF HistItem ;
METHOD Go (IN Integer) : REF HistItem ;
METHOD Append (IN REF HistItem ) ;
METHOD Delete : REF HistItem ;
METHOD Previous : REF HistItem ;
END Historic.

CLASS Historic IS ListWithPrev
IMPLEMENTS Historic ;
END Historic.
```

Le type et la classe de l'ensemble qui regroupe les versions créées à partir d'une autre version dans l'arbre de dérivation des versions d'un objet sont les suivants :

```
TYPE ListDerivItem IS ListWithPrev
METHOD First : REF DerivItem ;
METHOD Next : REF DerivItem ;
METHOD Last : REF DerivItem ;
METHOD Go (IN Integer) : REF DerivItem ;
METHOD Append (IN REF DerivItem ) ;
METHOD Delete : REF DerivItem ;
METHOD Previous : REF DerivItem ;
END ListDerivItem.

CLASS ListDerivItem IS ListWithPrev
IMPLEMENTS ListDerivItem ;
END ListDerivItem.
```

La classe qui réalise le type `VersManager` est la suivante :

```
CLASS VersManager IMPLEMENTS VersManager ;

SYNONYM DesigExp = String ;

historic : REF Historic ; derivTree : REF DerivItem ;

PROCEDURE LookUp (IN theObject : REF Top ; derivItem : REF DerivItem ;
                      desigExp : DesigExp) : Integer ;
// Procédure de recherche récursive d'une version à partir d'une expression
// de désignation. Il s'agit de l'implantation de la 'fonction interne' du
// GV. Son code est présenté dans [Guide-R8]
BEGIN
END LookUp ;

PROCEDURE AddUp (IN theObject : REF Top ; derivItem : REF DerivItem) : REF Bottom
// Procédure qui ajoute un nouvel élément à l'historique et un nouveau noeud
// à l'arbre de dérivation
position : Integer ; theCopy : REF Top ;
histItem, histItemNew : REF HistItem ;
derivItemNew : REF DerivItem ; listDerivItem : REF ListDerivItem ;
BEGIN
theCopy := theObject.Copy ;
// Copy est une méthode prédéfinie de la classe Top
IF theCopy = NIL
   THEN RETURN NIL ;
END ;
histItemNew := HistItem.New ;
derivItemNew := DerivItem.New ;
// Création du nouvel élément et du nouveau noeud
IF ((histItemNew = NIL) OR (derivItemNew = NIL))
   THEN RETURN NIL ;
END ;
IF historic # NIL
   THEN position := historic.NbItems ;
        histItem := historic.Last ;
        // Pour avoir un courant dans la liste de l'historique et pouvoir
        // ainsi ajouter un élément à la fin. Remarquons le suivant :
        // position = 0 sii histItem = NIL
   ELSE position := 0 ;
END ;
IF ((histItemNew.PutVersion(theCopy, derivItemNew) < 0) OR
    (derivItemNew.PutVersion(theCopy, position + 1, NIL, derivItem) < 0))
   THEN RETURN NIL ;
END ;
// Initialisation du nouvel élément et du nouveau noeud
IF derivItem # NIL
   THEN listDerivItem := derivItem.GetDaughters ;
        IF listDerivItem = NIL
           THEN listDerivItem := ListDerivItem.New ;
                IF listDerivItem = NIL
```

```
                    THEN theCopy := NIL ;
                    ELSE derivItem := listDerivItem.Last ;
                         // Pour avoir un courant dans la liste des filles d'une
                         // version dans l'arbre de dérivation et pouvoir ainsi
                         // ajouter un noeud à la fin
                END ;
            ELSE derivItem := listDerivItem.Last ;
                 // Pour avoir un courant dans la liste des filles d'une version
                 // dans l'arbre de dérivation et pouvoir ainsi ajouter un noeud
                 // à la fin
        END ;
END ;
IF theCopy = NIL
   THEN RETURN NIL ;
END ;
IF historic = NIL
   THEN historic := Historic.New ;
        IF historic = NIL
           THEN theCopy := NIL ;
           ELSE historic.Append(histItemNew) ;
                // addition d'un nouvel élément à l'historique
                IF derivTree = NIL
                   THEN derivTree := derivItemNew ;
                   ELSE listDerivItem.Append(derivItemNew) ;
                        // addition d'un nouveau noeud à l'arbre de dérivation
                END ;
        END ;
   ELSE historic.Append(histItemNew) ;
        // addition d'un nouvel élément à l'historique
        IF derivTree = NIL
           THEN derivTree := derivItemNew ;
           ELSE listDerivItem.Append(derivItemNew) ;
                // addition d'un nouveau noeud à l'arbre de dérivation
        END ;
END ;
RETURN theCopy ;
END AddUp ;

METHOD Init ;
BEGIN
historic := NIL ;
derivTree := NIL ;
END Init ;

METHOD CreateVers (IN theObject : REF Top ; desigExp : DesigExp) : REF Bottom ;
derivItem : REF DerivItem ;
BEGIN
IF theObject = NIL
   THEN RETURN NIL ;
END ;
derivItem := NIL ;
IF ((historic = NIL) AND (derivTree = NIL) AND
    ((desigExp.Length = 0) OR
```

```
      (desigExp = "self")   OR
      (desigExp = "first")  OR
      (desigExp = "last")))
   THEN RETURN AddUp(theObject, derivItem) ;
END ;
IF LookUp(theObject, derivItem, desigExp) < 0
   THEN RETURN NIL ;
END ;
IF derivItem = NIL
   THEN RETURN NIL ;
END ;
IF theObject # derivItem.GetObject
   THEN RETURN NIL ;
   ELSE RETURN AddUp(theObject, derivItem) ;
END ;
END CreateVers ;

METHOD DeleteVers (IN theObject : REF Top ; desigExp : DesigExp) : REF Bottom ;
METHOD GetVers (IN theObject : REF Top ; desigExp : DesigExp) : REF Bottom ;
// Le code de ces méthodes est présenté dans [Guide-R8]

END VersManager.
```

Le code des méthodes de gestion de versions qui se trouvent dans la classe `Top` est le suivant :

```
 . . .

versManager : REF VersManager ;
// Variable d'état qui comporte la référence-système d'un GV

 . . .

METHOD CreateVersion (IN desigExp : String) : REF Bottom ;
BEGIN
IF versManager = NIL
   THEN versManager := VersManager.New ;
        IF versManager = NIL
           THEN RETURN NIL ;
           ELSE versManager.Init ;
        END ;
END ;
RETURN versManager.CreateVers(SELF, desigExp) ;
END CreateVersion ;
```

```
METHOD DeleteVersion (IN desigExp : String) : REF Bottom ;
BEGIN
IF versManager = NIL
   THEN RETURN NIL ;
   ELSE RETURN versManager.DeleteVers(SELF, desigExp) ;
END ;
END DeleteVersion ;

METHOD GetVersion (IN desigExp : String) : REF Bottom ;
BEGIN
IF versManager = NIL
   THEN RETURN NIL ;
   ELSE RETURN versManager.GetVers(SELF, desigExp) ;
END ;
END GetVersion ;
```

AUTORISATION DE SOUTENANCE

DOCTORAT 3ème CYCLE, DOCTORAT INGENIEUR,
DOCTORAT DE L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

Vu les dispositions de l'Arrêté du **16 avril 1974**,

Vu les dispositions de l'Arrêté du **5 juillet 1984**,

Vu les rapports de M Michel ADIBA

M Marc SHAPIRO

M Rodrigo SCIOVILLE GARCIA est autorisé(e)
à présenter une thèse en vue de l'obtention du titre de DOCTEUR de
l'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

Grenoble, le 14 JUIN 1989

Le Président de l'Université
Joseph Fourier - Grenoble 1

Le Président,

A. NEMOZ

Résumé

Ce travail aborde quelques-uns des problèmes de la gestion d'objets dans un système informatique réparti.

GUIDE est un système d'exploitation expérimental réparti sur un réseau local. Les informations qu'il manipule sont structurées selon un modèle à objets. Le concepteur d'applications voit le noyau de ce système comme une machine virtuelle multi-sites et multi-processeurs dans laquelle le parallélisme est apparent et la distribution est cachée. Cette machine virtuelle réalise la gestion d'objets.

Ce travail décrit l'un des composants de la machine virtuelle : la mémoire d'objets. Il présente les choix de conception, sa réalisation et une analyse de l'expérience acquise. La mémoire d'objets est persistante. En effet, les objets ne sont pas détruits tant qu'ils restent liés à un objet particulier, persistant par définition. La mémoire d'objets est répartie et comporte deux niveaux : la Mémoire Virtuelle d'Objets, qui constitue le support d'exécution des objets, et la Mémoire Permanente d'Objets, qui se charge de leur conservation.

Certaines fonctions de gestion d'objets n'ont pas été intégrées dans la machine virtuelle mais elles sont mises en oeuvre comme des applications. On appelle ces applications les services du système. La conception et la réalisation des services de désignation symbolique et de gestion de versions sont décrites dans cette thèse.

Mots-clés

gestion d'objets, systèmes répartis, systèmes à objets, machine virtuelle, mémoire d'objets, mémoire persistante, désignation symbolique, gestion de versions