THESE

de l'Institut National Polytechnique

Présentée à

Grenoble

par

Daniel HAGIMONT

Adressage et protection dans un système réparti

Thèse soutenue devant la commission d'examen le :

19 octobre 1993

MM. Jean−Pierre Verjus	Président
Jean−Pierre Banâtre	Rapporteur
Elie Milgrom	Rapporteur
Jacques Mossière	Directeur de Thèse
Sacha Krakowiak	Examinateur

Je tiens à remercier

Monsieur Jean−Pierre Verjus, Directeur de l'Institut de Mathématiques Appliquées de Grenoble, qui m'a fait l'honneur de présider le jury de thèse,

Monsieur Jean−Pierre Banâtre, Directeur de l'Institut de Recherche en Informatique et Systèmes Aléatoires de Rennes, et Monsieur Elie Milgrom, Professeur à l'Université Catholique de Louvain, qui ont accepté d'être les rapporteurs de mon travail,

Monsieur Jacques Mossière, Professeur à l'Institut National Polytechnique de Grenoble, qui m'a encadré pendant ces trois années de labeur et plus particulièrement en phase de rédaction,

Monsieur Sacha Krakowiak, Professeur à l'Université Joseph Fourier et responsable du projet Guide, pour la confiance qu'il m'a accordée en m'accueillant dans son équipe.

Je tiens également à remercier toutes les personnes qui m'ont aidé et encouragé et plus particulièrement Xavier Rousset de Pina, Professeur à l'Institut National Polytechnique de Grenoble, pour son expertise sur bien des problèmes, ainsi que Cayu, Dédé, Pitch et Serge, sans qui Eliott ne serait pas né, sans oublier tous les membres de l'Unité Mixte Bull−IMAG.

Chapitre I

Introduction

Depuis une vingtaine d'années, les systèmes informatiques centralisés en temps partagé, qui constituaient le gros du marché de l'informatique, sont progressivement remplacés par des réseaux de stations de travail et de serveurs. Parmi les avantages d'une structure répartie, on peut citer notamment son coût XCrelativement faible, l'évolution possible des configurations par ajout de stations sur le réseau, la possibilité de diversifier les marques de matériels utilisés, ou celle de renouveler les machines progressivement.

Cette évolution des architectures matérielles s'accompagne d'une évolution des systèmes d'exploitation, la tendance étant à une intégration progressive. Une première étape a consisté à ajouter aux systèmes existants des primitives d'envoi et de réception de message entre les machines, puis un mécanisme d'appel de procédure à distance. Dans une seconde étape, il s'avère nécessaire d'intégrer aux systèmes d'exploitation des mécanismes plus élaborés et de fournir des outils d'écriture d'applications réparties. L'objectif est de gérer l'ensemble des machines du réseau comme s'il s'agissait d'une seule machine. Le système global est alors constitué des systèmes chargés sur toutes les machines du réseau, qui coopèrent pour tirer parti des diverses ressources physiques, processeurs et disques principalement. De nouveaux problèmes se posent alors, en particulier pour tirer parti de la répartition, pour assurer une meilleure disponibilité des données et une résistance aux défaillances.

Comme la plupart des concepteurs de logiciels, les concepteurs des systèmes répartis sont principalement soumis à deux influences : prise en compte des contraintes provenant de l'évolution des techniques et réponse aux besoins des usagers du système.

Les contraintes liées à l'évolution des techniques englobent tous les aspects intervenant dans la réalisation des services fournis par le système ; elles concernent à la fois les techniques de conception et le support matériel du système. Après une période où les programmeurs procédaient par extension à un système monolithique, en général Unix, les dernières années ont vu le retour en force de systèmes structurés en serveurs coopérants s'exécutant sur un micro−noyau. L'intérêt d'une telle approche est la modularité accrue du système, la simplification du portage (limité au micro−noyau) et l'extensibilité du système. Un autre exemple de contrainte concerne l'évolution des matériels utilisés pour constituer le système réparti. L'arrivée sur le marché de machines à grands espaces d'adressage (64 bits) et de réseaux rapides à hauts débits (de l'ordre du Gbit/s) va certainement modifier dans les années à venir les règles de conception de ces systèmes.

Les besoins des usagers évoluent vers un mode de travail caractérisé par la *coopération* autour d'une tâche commune à un ensemble d'individus, avec *partage* étroit

d'informations et communications en temps réel. Citons quelques exemples de telles applications coopératives :

- la gestion de documents, au sens large, y compris des données multimédia (image, etc), ou des ensembles de documents interconnectés (hypertextes),
- l'aide à la prise de décision dans un groupe (communication par courrier ou par panneaux d'affichage électroniques, gestion d'agendas, etc),
- l'ingénierie simultanée (gestion coopérative de projet, systèmes répartis pour la conception assistée),
- le développement de logiciel (gestion de versions et de configurations complexes).

Les applications coopératives se caractérisent par une interaction forte entre un ensemble d'utilisateurs, par l'intégration d'applications multiples et par le partage d'informations complexes et à longue durée de vie. Un des moyens de supporter des applications coopératives est d'utiliser la notion d'objet introduite dans le domaine des langages de programmation. L'intérêt d'utiliser la notion d'objet au niveau des langages de programmation réside dans la réponse apportée aux problèmes de l'ingénierie du logiciel, car elle impose un certain degré de structuration, de modularité et de concision. Pour le développement d'applications coopératives, l'objet peut être également utilisé comme unité de conservation et de partage de l'information. Les applications sont alors développées à l'aide de langages orientés−objets et le système doit fournir les mécanismes de base nécessaires aux compilateurs de ces langages. Pour permettre le partage entre les applications d'informations à longue durée de vie, le système doit assurer le partage et la persistance des objets, ce qui pose de nouveaux problèmes. Il devient notamment nécessaire de fournir des mécanismes de protection permettant aux usagers de contrôler les droits qu'ils accordent sur leurs propres objets aux autres usagers du système. En résumé, le système visé doit offrir un support pour des langages à base d'objets partagés persistants.

Répondre à ces problèmes est le but du projet Guide (Grenoble Universities Integrated Distributed Environment), mené à l'Unité mixte Bull−IMAG. Le projet Guide vise donc à concevoir et réaliser un système d'exploitation réparti, opérant sur un ensemble de stations de travail reliées par un réseau local, fournissant un support pour des langages à base d'objets et intégrant un modèle de communication basé sur le partage d'objets persistants. Les applications visées par un tel système sont des applications coopératives.

Dans une première phase du projet Guide, un prototype de système a été réalisé sur le système Unix. La réalisation de ce prototype a duré trois ans, de 1987 à 1989 et a donné à la fois naissance à un système et à un langage orienté−objet, appelé Guide, pour le développement d'applications réparties. Le choix de base à été de réaliser un noyau de système (Guide−1) spécifiquement conçu pour fournir le support nécessaire au langage à objets Guide. Cette première phase a permis de valider et d'écarter certains choix dans le modèle et sa conception. Ce prototype est encore utilisé pour expérimenter le développement d'applications coopératives réparties.

Dans une deuxième phase du projet, un prototype pré−industriel, qui sert de base au transfert vers un produit, a été réalisé (Guide−2). Ce nouveau prototype a profité de

l'expérience tirée de la première étude, mais il a également permis d'explorer des domaines qui n'ont pas été couverts par celle−ci, notamment :

- Le support de plusieurs langages orientés−objets

  Les langages visés sont le langage Guide et une extension du langage C++ gérant le partage et la persistance des objets.

- Des mécanismes de protection

  Dans Guide−1, les problèmes inhérents à la sécurité et à la protection ont volontairement (par manque de ressources) été laissés de côté. La nouvelle version de Guide a comme objectif d'assurer une certaine sécurité par isolation des applications et des utilisateurs et de fournir des mécanismes de protection permettant de contrôler les droits d'accès des utilisateurs aux objets.

Dans la conception de Guide−2, nous avons également voulu tirer parti de l'arrivée des micro−noyaux sur le marché . En effet, une des constatations principales de l'expérience sur Unix fut que le système Unix se prêtait mal à une réalisation efficace du modèle Guide. De plus, des expériences [Boyer91a][Freyssinet91a] réalisées sur les micro−noyaux Mach [Accetta86] et Chorus [Rozier88] ont montré que l'utilisation des micro−noyaux offrait des perspectives très intéressantes pour la réalisation de systèmes répartis à objets.

Mon travail de thèse se situe donc dans un projet dont l'objectif est la conception et la réalisation d'un système réparti à base d'objets partagés persistants, pour le développement et le support d'applications coopératives.

Dans ce but, j'ai défini un noyau de système appelé **Eliott** [Rousset93] qui fournit l'interface nécessaire aux compilateurs pour le support de langages orientés−objets. Cette interface prend la forme d'un modèle de base permettant le support de divers langages, ainsi que d'un ensemble de primitives permettant l'utilisation de ce modèle.

Le noyau Eliott, qui fournit ce modèle à objets de base, gère à la fois la persistance des objets et le partage des objets entre les structures d'exécution. Les objets sont regroupés en *grappes*, ce qui permet de les gérer plus efficacement dans le système ; ils peuvent être déplacés entre les grappes afin d'adapter dynamiquement les critères de regroupement. Les structures d'exécution sont des espaces d'adressage répartis, dans lesquels plusieurs flots d'exécution peuvent s'exécuter en concurrence. Les objets sont liés dynamiquement dans cet espace d'adressage. Pour assurer un niveau de protection minimum, le système isole les objets afin d'éviter qu'une erreur dans un programme puisse perturber d'autres objets. Un schéma d'adressage à la Multics a été défini : le premier appel à une méthode d'un objet est interprété et effectue les transformations nécessaires en vue d'une exécution efficace des appels suivants.

Des mécanismes de contrôle des droits d'accès aux objets sont intégrés à ce schéma d'adressage. Ils sont fondés sur des listes d'accès associées aux objets et définissant les droits accordés aux usagers du systèmes. Ces mécanismes permettent également la variation temporaire des droits d'un processus, ce qui permet la construction de sous−systèmes protégés mutuellement méfiants. La réalisation de ces mécanismes a été effectuée sans dégrader l'efficacité de l'appel de méthode. La vérification de la protection génére un coût

supplémentaire lors du premier appel qui est interprété, mais aucun coût pour les appels suivants.

Le noyau Eliott a été réalisée sur le micro−noyau Mach 3.0. Il sert actuellement de support au compilateur du langage Guide, qui a permis le développement de quelques applications ; ces applications ont permis d'effectuer des mesures sur le système et de valider nos choix de conception.

En résumé, les deux principaux apports de cette thèse sont :

- La conception et réalisation d'un noyau fournissant un support d'objets partagés persistants et servant de plate−forme pour des langages de programmation orientés−objets.
- La conception et l'intégration de mécanismes de protection dans le noyau de système réalisé, permettant la programmation d'applications protégées.

Ce rapport de thèse est organisé de la façon suivante.

Chapitre II

Le chapitre II est consacré à la position du problème. Il définit plus précisément les termes utilisés (objet, partage, persistance, répartition, protection), ainsi que les problèmes qui s'y rattachent dans le domaine étudié. Puis une modélisation du processus d'adressage des objets par les structures d'exécution est proposée, afin d'introduire la démarche utilisée pour présenter les différentes approches dans le chapitre III. Ce modèle s'appuie sur la distinction entre la mémoire de stockage, dont le rôle est d'assurer la persistance et de permettre la localisation des objets sur disque, et la mémoire d'exécution, dont le rôle est de permettre le partage et l'adressage des objets par les structures d'exécution des applications exécutées.

Chapitre III

Le chapitre III présente un état de l'art. Après une première partie consacrée à la présentation des systèmes utilisés pour illustrer cette étude, une classification des différentes approches est proposée, respectivement pour la mémoire de stockage et la mémoire d'exécution.

Chapitre IV

Le chapitre IV présente les motivations pour la conception du système Guide−2. La conception de la version actuelle du système Guide (Guide−2) a été précédée, donc influencée, par une première version (Guide−1). Ce chapitre présente donc cette première expérience, ainsi que son évaluation critique, qui a servi de base pour l'élaboration de la seconde version du système Guide.

Chapitre V

Le chapitre V présente les principes de conception du noyau Eliott. L'objectif de ce noyau est de fournir le support nécessaire pour la mise en œuvre de plusieurs langages orientés−objets et plus particulièrement les langages Guide et une extension

du langage C++ gérant le partage et la persistance des objets. Le noyau Eliott fournit un modèle à objet de base sur lequel des modèles plus complexes peuvent être construits. Il gère le partage et la persistance des objets et fournit un mécanisme d'appel de méthode efficace dans lequel les références aux objets ne sont interprétées qu'au premier appel.

Chapitre VI

Des mécanismes de protection permettant le contrôle de droits d'accès associés aux objets ont été intégrés au noyau Eliott. Pour présenter ces mécanismes, le chapitre VI étudie les différentes approches utilisées dans les systèmes d'exploitation, à savoir les listes d'accès et les capacités, puis il décrit notre proposition fondée sur des listes d'accès en montrant comment les problèmes classiques de protection peuvent être résolus.

Chapitre VII

La présentation de la réalisation du noyau Eliott sur le micro−noyau Mach 3.0 est proposée dans le chapitre VII. Une attention particulière est accordée aux mécanismes d'adressage et de protection des objets, car ils constituent les aspects originaux de mon travail.

Chapitre VIII

Le chapitre VIII propose une évaluation préliminaire du noyau Eliott. Après une description de l'état d'avancement des travaux (en juillet 1993), les applications utilisées pour cette évaluation sont décrites, puis les mesures effectuées sont commentées. Ces mesures ont deux objectifs : d'une part évaluer l'efficacité des mécanismes de base mis en œuvre par le noyau et d'autre part établir des données d'utilisation par les applications de ces mécanismes, afin de valider certaines des hypothèses ayant conduit aux choix de conception du système.

Chapitre IX

La conclusion résume les points essentiels de ce travail de thèse et propose une vision prospective du domaine des systèmes répartis à base d'objets partagés persistants.

Chapitre II

Objets partagés persistants dans les systèmes répartis : position du problème

Le but de ce chapitre est double :

- Expliquer ce qu'est une gestion d'objets partagés persistants dans un système réparti et préciser les motivations pour fournir un tel service. Dans ce but, nous définissons aussi clairement que possible ce que signifient les termes : **Objet**, **Partage**, **Persistance**, **Répartition**, **Protection**, puis nous présentons les problèmes posés par la gestion des objets par le système.
- Présenter un modèle du processus d'adressage des objets par les structures d'exécution afin de disposer d'une démarche d'analyse des différents systèmes que nous présentons au chapitre suivant.

II.1 Définitions

II.1.1 Objets

Le terme d'*objet* est généralement utilisé en informatique pour désigner toute unité d'encapsulation. Il peut être utilisé pour désigner un module contenant du code, une zone de données, ou même un processus serveur. S'agissant d'un domaine en évolution, les définitions rencontrées dans les différents projets de systèmes ne sont pas toujours en accord. Nous précisons donc notre propre définition des objets, proche de celle des langages de programmation.

Un objet est une entité contenant des données aussi appelées **état** de l'objet et exportant des opérations appelées **méthodes** qui permettent l'utilisation de l'objet. En général, l'état de l'objet est caché aux autres objets et la seule façon d'utiliser un objet est d'appeler une de ses méthodes. Cette définition est souvent associée au concept de **Classe**. La classe est un moyen donné au concepteur d'application pour définir une structure et un comportement commun à un ensemble d'objets. On dit que les objets qui ont la structure et le comportement décrit par une classe sont des exemplaires ou **instances** de cette classe. La définition d'une classe revient donc à définir :

- La structure des instances de la classe.

  Cette structure est commune à toutes les instances de cette classe. Les variables de cette structure sont aussi appelées **variables d'état** des instances.
- Le comportement des instances de la classe.

  Ce comportement est décrit dans la classe par la définition (le code) des méthodes appelables sur ses instances.

Pour permettre une programmation plus aisée et de façon incrémentale (par réutilisation de classes existantes), un mécanisme appelé **héritage** est également souvent fourni. L'héritage est un mécanisme permettant de définir une nouvelle classe à partir d'une autre, en décrivant simplement en quoi elle diffère de la première. On peut alors enrichir la définition de la nouvelle classe en ajoutant de nouvelles variables d'état, en décrivant de nouvelles méthodes (l'interface et le code de ces méthodes), ou en redéfinissant des méthodes. On dit que la classe redéfinie est **sous−classe** de la première classe, qui elle, est **super−classe** de la seconde. Une classe peut être la super−classe de plusieurs classes. Elle décrit alors un comportement commun à toutes ses sous−classes. L'héritage est dit **simple** si une classe ne peut avoir qu'une seule super−classe. Dans le cas contraire, on est en présence d'héritage **multiple**.

La notion d'objet est généralement fournie par un langage de programmation. Dans ce domaine [Chin91], on dit parfois qu'un langage de programmation est "basé−objet" s'il offre la notion d'objet et de classe et "orienté−objet" s'il offre en plus la notion d'héritage.

Le rôle d'un langage de programmation est de permettre l'écriture de programmes ; le compilateur de ce langage traduit les programmes en code réalisant des appels à une machine virtuelle fournie par le système d'exploitation de la machine. La machine virtuelle réalisée par un système fournit des abstractions pouvant être de plus ou moins haut niveau. Dans le cas d'une machine−langage, les abstractions fournies par le système sont de haut niveau et visent au support d'un langage unique. Si le système permet le support de plusieurs langages, il doit alors fournir des abstractions qui sont généralement plus élémentaires.

Le succès des langages orientés−objets, dû à la modularité et aux possibilités de réutilisations qu'ils apportent, a poussé les concepteurs de systèmes à fournir des supports efficaces pour la mise en œuvre de ces langages. Ces systèmes proposent en général l'objet comme unité d'accès, de conservation et de partage de l'information. Etant donné que les langages orientés−objets ont souvent des conceptions différentes de l'héritage, ces systèmes ne fournissent généralement pas de support pour l'héritage. Nous nous limitons ici volontairement à l'étude du support par le système de langages "basés−objets".

II.1.2 Partage

Pour le développement d'applications coopératives, l'intérêt d'un modèle de programmation fondé sur le partage implicite des données manipulées est de fournir aux utilisateurs une abstraction très commode et notamment de masquer les mécanismes de communication sous−jacents. En effet, au niveau des langages, le partage implicite de données permet une programmation plus aisée par rapport à l'utilisation explicite de primitives d'envoi de message, ou de recopie dans une zone partagée. Le développeur d'application peut programmer

sans se soucier du fait que les données manipulées doivent être partagées, bien qu'il doive ensuite synchroniser l'accès à ces données.

Puisque nous évoluons dans le domaine des applications coopératives développées à l'aide de langages à objets, nous visons donc le partage implicite d'objets entre les structures d'exécution. Nous disons qu'il y a partage d'objets entre des processus différents dès que ces processus ont la possibilité d'appeler les méthodes d'un même objet. Permettre le partage d'objets, c'est tout d'abord fournir un moyen de partager des identificateurs d'objets pour les désigner, puis permettre à tout processus d'utiliser un objet (par appel de méthode) à partir de son identification.

De nombreux systèmes, sur lesquels des langages orientés−objets existent, n'offrent pas la notion de partage d'objets. Les seuls moyens de partage offerts sont la recopie de données dans une zone partagée (en mémoire virtuelle ou sur disque) ou l'envoi de messages, dont l'utilisation peut s'avérer fastidieuse (et coûteuse) pour des structures d'objets complexes se référençant entre eux. C'est le cas notamment du système Unix sur lequel on peut écrire des programmes en langage C++ ; il ne permet pas le partage implicite des objets créés par les applications, ces objets étant créés dans la mémoire virtuelle des processus utilisateurs.

Le partage nécessite donc la gestion d'un espace de **noms d'objets** partageable entre applications, ainsi que des mécanismes de **partage** physique de la mémoire entre les applications. Un objet doit pouvoir contenir des noms d'objets, ce qui permet de construire des structures complexes.

Dans notre cas, nous supposons que cet espace de noms d'objets géré par le système a comme unique but l'identification et le partage d'objets. Un service de nommage symbolique, permettant d'associer des chaînes de caractères à des noms d'objets, peut être réalisé comme une application du système que nous définissons.

II.1.3 Persistance

Dans la plupart des systèmes traditionnels, la gestion de la sauvegarde des données était laissée à la charge du programmeur d'applications, ses outils étant principalement les primitives du système de gestion de fichiers. De plus, cette gestion de la persistance n'était pas uniforme, puisque les données persistantes et temporaires n'étaient pas désignées de la même façon.

La volonté d'uniformiser l'accès à ces deux types de données et de décharger le programmeur de la tâche fastidieuse de sauvegarde a donné naissance à des systèmes où la gestion de la persistance des objets est implicite. Ces systèmes permettent alors de définir des objets temporaires ou persistants, tous ces objets étant désignés de façon uniforme.

II.1.4 Répartition

Gérer la répartition, c'est permettre l'utilisation de toutes les ressources disponibles sur le réseau de stations de travail. Le fait que les ressources soient réparties signifie que les objets manipulés dans le système sont disséminés sur tous les disques des machines. Il faut donc être capable de **localiser** un objet dans le système réparti et permettre son partage entre des applications ne s'exécutant pas forcément sur la même machine.

Pour offrir une gestion d'objets efficace, il faut à la fois permettre le regroupement de certains objets pour augmenter la localité d'accès et permettre la répartition de ces objets afin d'augmenter le parallélisme de l'exécution.

Enfin, intégrer la répartition, c'est également ne plus la montrer, ou plutôt ne la montrer que lorsque c'est nécessaire. Le nommage des objets ne doit donc en aucun cas être lié à des noms de machines. Il doit donner la vision d'un espace uniforme d'objet, cachant le fait que ces objets sont répartis dans le réseau.

II.1.5 Protection

Un des concepts clés associé à la notion d'objet est l'encapsulation des données. Cela signifie que la seule façon d'accéder en lecture ou en écriture à l'état d'un objet est l'appel à une méthode de cet objet.

Cette encapsulation peut être assurée au niveau des langages de programmation utilisés, lorsque ces langages sont fortement typés. Un langage est fortement typé si chaque variable se voit associer un type, qui définit l'ensemble des opérations permettant la manipulation d'une variable de ce type. Le compilateur du langage peut alors assurer l'encapsulation des objets, grâce à des contrôles statiques (réalisés lors de la compilation) et à des contrôles dynamiques (réalisés à l'exécution), en vérifiant que seuls des opérateurs autorisés sont appliqués aux variables. Ces langages ne permettent généralement pas la manipulation directe d'adresses en espace virtuel et n'offrent que la manipulation de types de base et de références à des objets. Les débordements de tableaux sont contrôlés et l'utilisation d'une référence à un objet n'est possible que par appel de méthode.

D'autres langages, comme C++, permettent d'adresser toutes les données de l'espace virtuel courant en forgeant des pointeurs et n'assurent donc pas cette encapsulation. Ce manquement à la sécurité n'est pas très grave dans un contexte où les applications ne partagent pas d'objets, une erreur dans un programme ne pouvant détruire que des objets de la même application, donc du même utilisateur.

Dès l'instant où l'on permet aux applications de partager des objets, le problème de l'encapsulation des objets devient critique, l'appel d'une méthode sur un objet appartenant à une autre application (resp. un autre utilisateur) ne signifiant pas forcément une confiance absolue envers cette application (resp. cet utilisateur). Il faut qu'un appel de méthode ne rende pas l'appelant ou l'appelé sensible aux erreurs ou malveillances de son alter−ego. Nous appelons **confinement** le fait de garantir un certain degré d'isolation entre des applications et des usagers pouvant partager des objets. Lorsqu'il n'est pas réalisé au niveau des langages de programmation, ce confinement doit être assuré par le système. Il peut alors assurer l'isolation de chaque objet ou l'isolation d'ensembles d'objets se faisant mutuellement confiance.

De plus, même si l'encapsulation des objets est garantie par le système et ne permet que l'appel de méthode sur les objets partagés, il faut permettre au programmeur d'application de contrôler les droits des utilisateurs en terme de méthodes appelables sur les objets partagés. Nous appelons **contrôle d'accès** le mécanisme permettant de limiter la vue (en terme de méthodes appelables) d'un utilisateur sur un objet.

II.2 Processus d'adressage

Nous allons maintenant étudier les problèmes qui se posent lorsque l'on veut fournir un système offrant les concepts définis plus haut (partage d'objets, répartition, persistance, protection) et plus particulièrement dans le processus d'adressage des objets.

Pour cette étude, nous modélisons le processus d'adressage d'un objet de la façon suivante.

Nous appelons **activité** un flot d'exécution séquentiel dont le comportement consiste à exécuter une suite d'appels de méthode sur des objets. Une application peut être composée de plusieurs activités s'exécutant en parallèle, sur une même machine ou sur plusieurs machines.

Nous appelons **mémoire d'exécution** le niveau de mémoire dans lequel un objet doit être chargé pour pouvoir être adressé par un processeur (généralement au moyen d'une adresse virtuelle). Cette mémoire d'exécution est donc composée à un instant donné de l'ensemble des espaces de pagination existant sur l'ensemble des stations de travail du réseau.

Nous appelons **mémoire de stockage** le niveau de mémoire contenant les objets de façon persistante. Elle est constituée par les disques des machines du réseau.

Les applications sont composées d'un ensemble de classes. Au niveau des langages de programmation, des variables de type **référence objet** peuvent figurer dans la pile ou dans l'état d'un objet. Ces variables servent à identifier des objets et contiennent a priori des noms d'objet.

Dans ces deux niveaux de mémoire, les objets sont désignés par des noms. Nous supposons que le nommage des objets est uniforme, c'est à dire que tous les noms d'objets ont le même format, que l'objet soit en espace d'exécution ou de stockage. Nous appelons **résolution de nom** l'action qui permet d'obtenir la localisation d'un objet à partir de son nom. Cette résolution peut intervenir à la fois en mémoire d'exécution et en mémoire de stockage. Dans le processus d'adressage, une activité qui utilise un nom d'objet pour appeler une méthode sur cet objet va tout d'abord essayer de résoudre ce nom dans la mémoire d'exécution. Si cette résolution réussit, alors l'activité peut utiliser l'objet. Dans le cas contraire, une résolution du nom en mémoire de stockage doit avoir lieu pour retrouver l'objet sur disque. Puis cet objet doit être rendu disponible en mémoire d'exécution. On dit généralement qu'il y a eu **défaut d'objet** en mémoire d'exécution.

On peut donc caractériser le processus d'adressage dans un système à base d'objets par :

- la gestion de la mémoire d'exécution,
- la gestion de la mémoire de stockage,
- la résolution des noms dans ces deux espaces.

Nous présentons tout d'abord la nature des noms utilisés dans un système (section II.2.1), puis nous rappelons en quoi consiste l'opération de résolution de nom (section II.2.2). Nous présentons ensuite des modèles de mémoire d'exécution et de mémoire de stockage, ainsi que les motivations pour leur réalisation (sections II.2.3 et II.2.4).

II.2.1 Nature des noms

Dans un système, on peut trouver différents types de noms [Legatheaux88] :

- Des noms symboliques.

  Les noms symboliques, qui sont en général des chaînes de caractères, sont utilisés au niveau utilisateur. Un espace structuré en répertoires est souvent construit. Dans notre cas, nous supposons que ces noms sont gérés à un niveau supérieur, comme une application du système que nous réalisons.

- Des noms d'objets internes au système.

  La gestion de ce nommage est notre objectif principal. Ces noms permettent de partager des objets par appel de méthode en désignant l'objet appelé par ce nom.

- Des adresses virtuelles.

  Ce sont les noms utilisés par les processeurs pour désigner des objets. La résolution de nom en mémoire d'exécution permet d'obtenir une adresse virtuelle à partir d'un nom interne d'objet.

- Des adresses disques.

  Ce sont les noms utilisés pour désigner des objets sur les disques. La résolution de nom en mémoire de stockage permet d'obtenir à partir d'un nom interne d'objet une adresse disque.

II.2.2 Résolution de noms

Dans son étude des systèmes de nommage et de liaison, J. H. Saltzer [Saltzer78] modélise un système de nommage d'objets de la façon suivante :

- Un **contexte** est une entité qui enregistre des associations entre des noms et des objets. Un nom n'a donc une signification que par rapport à un contexte.
- Une opération appelée **liaison** permet d'ajouter une association entre un nom et un objet dans un contexte. On dit que le nom est lié à l'objet dans ce contexte.
- Une opération appelée **résolution de nom** permet d'interroger un contexte pour localiser un objet à partir de son nom.

En fait, des contextes de résolution de noms existent à tous les niveaux dans un système. Dans notre modèle, ils peuvent intervenir en mémoire de stockage comme en mémoire d'exécution.

On peut en général distinguer les contextes globaux des contextes locaux :

- Les contextes globaux donnent une traduction universelle aux noms qu'ils définissent. On dit que les noms sont globaux ou absolus. Un nom global a donc le même sens dans toute la mémoire (de stockage ou d'exécution).
- Les contextes locaux permettent de ne pas avoir la même traduction d'un nom dans tout le système. Les contextes locaux sont utilisés en général pour gérer des noms relatifs à une partie de la mémoire. La mémoire est découpée en parties et un même nom peut désigner des objets différents dans des parties différentes.

II.2.3 Mémoire d'exécution

Nous allons tout d'abord décrire les deux modèles d'exécution classiques qui sont le modèle à objets actifs et le modèle à objets passifs, le choix d'un modèle étant indépendant des réalisations possibles décrites dans le chapitre suivant. Puis nous présentons les motivations pour la réalisation de la mémoire d'exécution.

II.2.3.1 Modèles d'exécution

Lorsqu'un système fournit un support pour des langages à base d'objets, on distingue deux façons d'envisager l'utilisation d'un objet par une activité, connues sous les noms de **modèle à objets actifs** et **modèle à objets passifs** :

- Dans un modèle à objets actifs, un objet est composé de son état et d'un certain nombre d'activités. Ces activités restent associées à cet objet et servent des requêtes d'exécution de méthode pour cet objet qui réagit comme un serveur. Lorsqu'une activité dans un objet (qui exécute une méthode sur cette objet) doit appeler une méthode sur un autre objet, elle envoie une requête à cet objet et suspend son exécution. Les activités qui ne sont pas utilisées dans l'objet appelé attendent les requêtes et y répondent en exécutant l'appel de méthode demandé. A la fin de l'appel de méthode, une réponse est envoyée à l'activité dans l'objet appelant qui reprend alors son exécution. Dans un tel schéma, les activités associées à un objet peuvent être allouées statiquement ou dynamiquement. Si l'allocation des activités est statique, les requêtes sont traitées par les activités serveurs et mises dans une file d'attente si toutes les activités sont occupées. Si l'allocation est dynamique, une nouvelle activité est créée pour traiter chaque requête et détruite lorsque la requête est traitée.

  Des variantes de ces deux options existent. Une réalisation classique du modèle actif consiste à associer un processus au sens Unix du terme à chaque objet, les envois de requêtes étant réalisés par envoi de message entre les processus. Une telle réalisation n'est pas très judicieuse, car elle impose une grosse taille d'objets (de l'ordre de grandeur d'un espace virtuel).

- Dans un modèle à objets passifs, un objet est composé uniquement de son état. Les activités qui exécutent des méthodes sur ces objets sont des entités complètement séparées des objets. Une activité exécute un appel de méthode en *couplant* l'objet appelé dans son espace d'adressage. L'opération de couplage(1) revient à modifier l'espace d'adressage de l'activité concernée pour que celle−ci puisse adresser l'objet à appeler (par une adresse virtuelle).

  Une réalisation classique de ce mécanisme de couplage est l'opération d'attachement (*attach*) de mémoire partagée d'Unix, si l'activité est réalisée par un processus Unix. Le problème de taille des objets décrit ci−dessus se pose ici aussi, car un segment de mémoire partagée d'Unix est composé d'un ensemble de pages, ce qui impose une grosse taille d'objet.

La figure Fig. 2.1 illustre de façon comparative les deux modèles d'exécution.

---

(1) Cette notion est plus précisément définie au chapitre suivant.

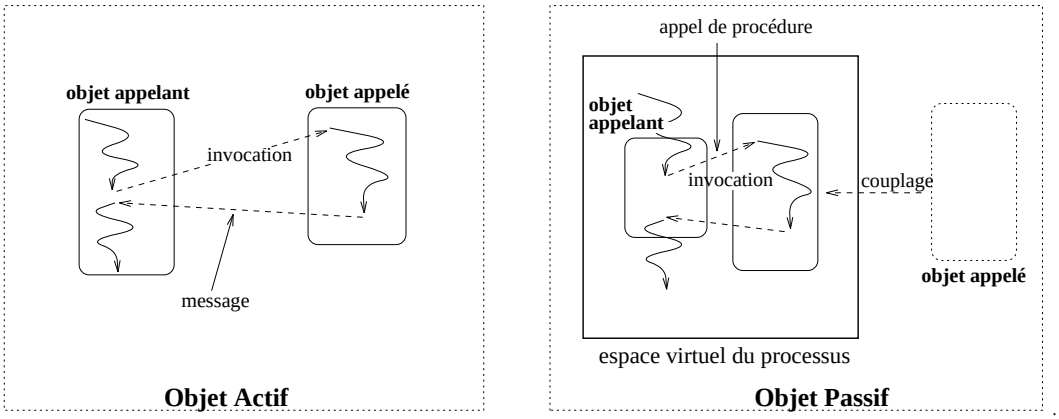

Fig. 2.1 : Modèles à objets actifs et passifs

II.2.3.2 Motivations pour la réalisation

Dans les deux cas, le but à atteindre est de permettre aux activités, s'exécutant de façon concurrente sur une machine ou sur des machines différentes, de partager des objets en mémoire d'exécution (en appelant simultanément une méthode sur le même objet).

La réalisation du partage doit toutefois tenir compte du degré de confinement désiré. S'il est dans certains cas acceptable de partager tous les objets entre toutes les activités sans aucune protection, ce n'est en général pas le cas.

Enfin, il faut permettre la résolution des noms (ou localisation) des objets en mémoire d'exécution. Nous appelons cette opération "adressage" des objets en mémoire d'exécution. L'adressage des objets doit être efficace et le système ne doit pas faire payer le partage et la persistance des objets à un programme ne les utilisant pas.

En résumé, il faut **adresser** efficacement des objets **partagés** de façon **protégée**.

II.2.4 Mémoire de stockage

II.2.4.1 Modèle de la mémoire de stockage

La mémoire de stockage est répartie sur l'ensemble des stations de travail connectées au réseau. Pour des raisons matérielles (les disques) ou de structuration logique, la mémoire de stockage est généralement composée de sous ensembles d'objets appelés **partitions**.

II.2.4.2 Motivations pour la réalisation

La gestion de la mémoire de stockage doit mettre en œuvre, outre les allocations et libérations de l'espace mémoire, la résolution des noms (ou localisation) des objets en mémoire de stockage, permettant de retrouver l'objet dans les partitions réalisant ce stockage. Nous appelons cette opération "localisation" des objets en mémoire de stockage.

L'unité de transfert entre la mémoire de stockage et la mémoire d'exécution peut être l'objet, la partition, ou une unité intermédiaire (la page, un sous regroupement logique). Le

regroupement d'objets dans une telle unité est donc un critère important pour les performances du système, car il permet notamment d'améliorer l'efficacité des opérations d'entrée/sortie. La gestion de la mémoire de stockage doit permettre la migration des objets entre partitions (donc entre sites), permettant de regrouper des objets liés dans le cadre d'une application.

En résumé, les principales fonctions auxquelles nous nous intéressons dans le cadre de cette étude sont la **localisation** et la **migration** des objets gérés dans un espace composé de partitions réparties sur le réseau.

II.3 Conclusion

Nous avons défini dans ce chapitre en quoi consistait une gestion d'objets partagés persistants dans un système réparti. Nous avons modélisé le processus d'adressage d'un objet qui se traduit par la gestion de deux niveaux de mémoire :

- la mémoire d'exécution qui supporte la gestion d'activités appelant des méthodes sur les objets,
- la mémoire de stockage qui fournit le support pour la persistance des objets.

Le processus d'adressage d'un objet intervient lorsqu'une activité appelle une méthode sur un objet. L'activité essaie de localiser l'objet en mémoire d'exécution. Si cette localisation échoue, il y a défaut d'objet en mémoire d'exécution. L'objet est alors localisé en mémoire de stockage et amené en mémoire d'exécution. L'appel effectif de la méthode peut alors avoir lieu.

Les propriétés que doit assurer la gestion de la mémoire d'exécution sont le partage, la protection (confinement) et l'efficacité de l'adressage. La mémoire de stockage doit permettre la localisation des objets, les objets pouvant être déplacés entre les partitions de stockage gérées.

Les gestions de la mémoire d'exécution et de la mémoire de stockage doivent être coordonnées pour être efficaces. En principe, des contextes de résolution de noms doivent être gérés dans ces deux espaces pour y localiser les objets, mais le choix de la technique de nommage des objets peut permettre de simplifier la gestion des certains contextes. Les différentes techniques utilisées sont décrites dans le chapitre suivant et illustrées par des exemples de projets de recherche dans ce domaine.

Chapitre III

Objets partagés persistants dans les systèmes répartis : différentes approches

De nombreuses expériences ont été menées dans ce domaine. Le but de ce chapitre n'est donc pas de présenter tous les projets de recherche traitant ce sujet, mais plutôt de classifier les différentes approches en les illustrant à travers des projets pertinents. Nous allons tout d'abord présenter rapidement les systèmes qui sont utilisés pour illustrer cette étude, puis nous proposerons une classification des différentes approches en détaillant les caractéristiques des systèmes cités en exemple.

III.1 Systèmes étudiés

III.1.1 Argus

Argus [Liskov85] est un projet de recherche mené au Massachusetts Institute of Technology (MIT) de 1983 à 1987. L'objectif d'Argus est de concevoir à la fois un système et un langage de programmation, conçus pour le développement d'applications réparties gérant des données persistantes et permettant un accès concurrent à ces données.

Les deux principales abstractions définies par Argus sont le *Guardian*(1) et l'*Action :*

- Un *Guardian* est un objet actif désigné de façon unique dans le système.

  Un *Guardian* encapsule un ensemble d'objets de plus petite taille et un ensemble de flots d'exécution (*threads*) pouvant opérer sur ses objets. Les *threads* dans un *Guardian* sont créés à la demande. L'interface d'accès à un *Guardian* est composée d'un ensemble de méthodes (*handler*), permettant les appels entre *Guardians*. Les objets locaux ne peuvent pas être partagés entre les *Guardians*, ils n'appartiennent qu'à un seul *Guardian*. Les *Guardians* peuvent être répartis sur les machines du réseau.
- Une *Action* est une transaction.

  Les *Action*s sont sérialisées et atomiques (réussies ou annulées), et elles peuvent être imbriquées et exécutées de façon concurrente.

---

(1) On gardera systématiquement les terminologies des systèmes étudiés.

Argus a été réalisé [Liskov87] sur le système Unix, sur un réseau de Micro VAX−II connectés par un réseau Ethernet. Les *Guardians* sont réalisés par des processus Unix et les appels de *Handlers* sont réalisés par échanges de message entre les processus Unix. Le système Argus est présent dans chaque processus et coordonne la coopération des *Guardians*.

III.1.2 Thor

Thor [Liskov92] est également un projet de recherche mené au MIT depuis 1992. Le but de Thor est de fournir un système gérant des objets pour des applications réparties de type base de données.

Comme Argus, Thor permet le partage d'objets à l'aide de transactions. Par contre, il diffère de celui−ci par le fait que les objets sont passifs. Les principales abstractions proposées par Thor sont :

- Les objets. Les objets de Thor [Day92] sont regroupés en espace de stockage dans des entités appelées *Object Repositories* (*ORs*) et peuvent migrer entre les *ORs*. Un modèle à objets de base est fourni pour le support de plusieurs langages de programmation.
- Les processus. Chaque processus de Thor est confiné dans un espace d'adressage appelé *Front End* (*FE*). Les *FEs* coopèrent avec les *ORs* pour charger les objets et le code des méthodes.

L'objectif de Thor est d'avoir des machines spécialisées pour le stockage (contenant les *ORs*) et des machines utilisateurs (supportant les *FEs*).

L'originalité de Thor réside dans la technique utilisée pour la désignation des objets, qui permet de résoudre facilement le problème de la migration des objets.

Un premier prototype a été développé et programmé en Argus.

III.1.3 Clouds

Clouds [Dasgupta90][Dasgupta91] est un projet de recherche du Georgia Institute of Technology dont le but est d'explorer la conception et la réalisation d'un système distribué à base d'objets sur un réseau de stations de travail.

Le modèle proposé par Clouds est fondé sur deux entités : l'*objet* et le *thread* :

- Les objets dans Clouds sont des objets passifs, utilisés par appel de méthode par des *threads*, le *thread* étant l'unité d'exécution. Les objets sont implicitement persistants et nommés par une référence unique allouée par le système. La localisation des objets est cachée à l'utilisateur et gérée par le système.
- Un *thread* de Clouds est un processus exécutant des appels de méthode dans le système distribué ; il peut changer de machine d'exécution pour des raisons de localité d'accès aux objets. Les *threads* peuvent partager des objets, ce qui permet la coopération entre les applications. Plusieurs *threads* peuvent s'exécuter en parallèle dans un objet et un objet peut être partagé par des *threads* sur des machines différentes.

Le projet a débuté en 1986 et a donné naissance à une première version (*Clouds v.1*) développée sur un ensemble de Vax−11/750. Dès le début du projet, il a été décidé que Clouds serait un système natif, c'est à dire développé sur une machine nue, sans bénéficier d'un noyau ou système préexistant. Pour améliorer la modularité du système et permettre son extension, une nouvelle version a été développée (sur des stations Sun 3/60). Cette version est composée d'un noyau minimal appelé *Ra* et de la version *Clouds v.2* sur le noyau *Ra* [Bernabeu88].

III.1.4 Orca

Orca [Bal92] est un langage pour le développement d'applications parallèles conçu à l'Université Libre d'Amsterdam. Le but d'Orca est de permettre le partage d'objets entre des flots d'exécution parallèles s'exécutant sur un réseau de stations de travail.

Orca fournit les notions d'objets passifs et de processus séquentiels. Le mode de partage d'objets entre processus est l'héritage : un processus peut créer un processus fils et permettre à ce processus de partager certains objets avec lui. Ainsi, une application peut s'étendre en créant des processus distribués partageant des objets. La persistance n'est pas un objectif d'Orca. L'originalité d'Orca réside dans la technique utilisée pour le partage d'objets entre des processus s'exécutant sur des machines différentes.

Le projet a mené à la réalisation d'un prototype [Bal89] sur le système Amoeba [Mullender86], développé au même endroit.

III.1.5 Emerald

Emerald [Black86] est un projet de recherche mené à l'Université de Washington, par une partie de l'équipe du projet Eden [Black85]. Si dans Eden, la priorité avait été donnée à la définition du système, le principal objectif d'Emerald est la définition d'un langage pour la programmation d'applications réparties, ainsi que la réalisation d'un compilateur de ce langage et d'un noyau d'exécution.

Dans Emerald, il n'y a qu'un seul type d'objet, mais le compilateur choisit le mode de représentation de chaque objet parmi les trois suivants :

- Les objets directs représentant les types de bases (entiers, caractères ...).
- Les objets locaux dont la visibilité est limitée à l'objet englobant et sur lesquels les appels de méthode peuvent être réalisés par simple appel de procédure.
- Les objets globaux qui sont désignables dans tout le système, qui sont actifs et qui peuvent contenir des objets locaux et référencer d'autres objets globaux. Les objets globaux communiquent par des appels de méthode réalisés par échanges de messages.

La seule structure d'exécution proposée par Emerald est la création d'un processus optionnel créé lors de l'initialisation d'un objet global. Ce processus peut naturellement appeler d'autres objets, la concurrence dans un objet global résultant de l'exécution de plusieurs processus optionnels.

Le système Emerald a été réalisé sur le système Unix.

III.1.6 Amber

Amber [Chase89] est un système distribué à base d'objets réalisé à l'université de Washington. Le but du projet Amber est d'offrir un environnement pour l'écriture d'applications parallèles. Les applications d'Amber sont écrites en utilisant un sous−ensemble du langage C++. Les objets sont passifs, ils ne sont ni persistants, ni partageables entre les applications.

Une application dans le système Amber consiste en un espace virtuel d'objets, potentiellement réparti sur plusieurs stations de travail. Ainsi, une application peut exister sur plusieurs sites et être constituée de plusieurs espaces virtuels locaux. Un objet est toujours créé localement (sur le site courant), dans le représentant local de l'espace virtuel réparti. Une primitive (*MoveTo*) permet à l'utilisateur de faire changer le site de résidence d'un objet de façon explicite.

Le programmeur d'application a la possibilité de créer dans une même application des flots d'exécution appelés *threads*, qui peuvent s'exécuter de façon concurrente sur un même processeur, sur plusieurs processeurs d'un multiprocesseur, ou sur plusieurs stations de travail.

Un *thread* ne peut adresser un objet que localement, ce qui implique que lorsque l'objet et le *thread* sont distants, un appel de méthode à distance est utilisé. Lorsqu'une migration d'objet est demandée par l'utilisateur, les activités en cours d'exécution sur cet objet migrent également. Il est également possible d'attacher (*Attach*/*Unattach*) des objets entre eux, ce qui signifie que la migration d'un objet implique la migration de tous les objets qui lui sont attachés. Il est possible de définir des objets en lecture seule, qui sont alors dupliqués sur tous les sites où l'objet est demandé.

Cet environnement a été développé en 1988−1989 sur un réseau de stations de travail multiprocesseurs FireFly construites par DEC.

III.1.7 Opal

Opal [Chase92a] est un projet de recherche mené à l'Université de Washington dont le but est d'explorer l'influence de l'arrivée des grands espaces d'adressage sur la conception des systèmes répartis à base d'objets [Chase92b][Chase92c]. L'idée de base est que l'arrivée de processeurs dont l'espace virtuel est adressable par une adresse de 64 bits doit permettre une réalisation plus efficace du support des objets.

Ce projet, qui a débuté en 1991−1992, constitue la suite du projet Amber mené par la même équipe. Contrairement au projet Amber, Opal offre la notion d'objets persistants partagés entre les applications.

Le but d'Opal n'est pas de fournir directement une gestion d'objets au niveau du système, mais de fournir un support système adapté à la gestion d'objets à un niveau supérieur.

Le modèle proposé par Opal repose sur quatre entités : le *thread*, le *segment*, le *domaine de protection* et le *port* :

- Le *thread* est l'unité d'exécution d'Opal. Une application peut être composée de plusieurs *threads* s'exécutant en parallèle sur une ou plusieurs machines.

- Le *segment* est l'unité de partage et de stockage. Un segment est désigné par une capacité (un identificateur protégé que l'on ne peut se forger), il peut contenir des données ou du code.
- Un *domaine de protection* est un ensemble de droits d'accès à des segments, donc un ensemble de capacités de segments. Un *thread* s'exécute à un instant donné dans un domaine et il ne peut adresser un segment que si le domaine possède une capacité sur ce segment.
- Un *port* est un point d'entrée dans un domaine de protection. Un port appartient à un domaine et permet, par un appel système, de transférer l'exécution du *thread* courant vers un autre domaine de protection.

Un modèle à objets est conçu à un niveau supérieur. Conceptuellement, les objets d'Opal sont des objets passifs. Un *thread* couple un objet dans l'espace d'adressage de son domaine de protection pour y accéder. Le partage d'objets par plusieurs *threads* dans un même domaine ou dans des domaines différents permet la communication entre ces *threads.*

Une application est composée de domaines de protection et de *threads*. Pour offrir une protection sur l'accès à des objets, une application peut interdire le couplage de certains segments à des domaines d'autres applications et transmettre (par partage) une identification de port. Ainsi, les autres applications pourront appeler des méthodes sur les objets de la première par changement de domaine de protection, mais ne pourront pas y accéder par couplage. Pour offrir une protection au niveau de l'objet, Opal propose l'utilisation de *pointeurs protégés* : un pointeur protégé est composé d'une désignation de port, d'une désignation d'objet et d'une clé complexe difficile à contrefaire. Un appel de méthode protégé peut alors être réalisé par transfert vers un domaine de protection protégé en utilisant un port et en donnant une clé pour s'authentifier auprès de ce serveur.

Le système Opal est actuellement en cours de développement sur le micro−noyau Mach 3.0.

III.1.8 Gothic

Gothic [Banâtre91][Puaut93] est un projet de recherche mené à l'IRISA de Rennes, dont un des objectifs est l'étude et la mise en œuvre d'un système fournissant le support nécessaire à un langage orienté−objet appelé Arche, dans un environnement réparti.

Les objets dans Gothic sont persistants. Ils peuvent être répartis sur l'ensemble des disques du réseau de machines, cette répartition étant masquée à l'utilisateur du système. Il est possible de faire migrer un objet d'un site de stockage à un autre.

A l'exécution, les objets de Gothic sont des entités actives. Plusieurs flots d'exécution peuvent s'exécuter en concurrence dans un objet. Un placement dynamique des calculs sur les machines du réseau est réalisé.

Un prototype a été développé sur le micro−noyau Mach 3.0 et supporte des applications réparties écrites en langage Arche.

III.1.9 Guide−1

Guide−1 [Balter91] est le premier prototype qui a été développé sur le système Unix dans le cadre du projet Guide, de 1987 à 1990. Les choix de base pour la conception de Guide−1 sont principalement :

- La réalisation d'une machine virtuelle offrant les services essentiels pour le support d'un langage unique (le langage Guide [Krakowiak90]).
- Des structures d'exécution appelées *Domaines* contenant des flots d'exécution appelés *Activités.*

  Un domaine est un espace d'adressage contenant des objets accessibles par des activités. Les activités sont des flots séquentiels d'exécution concurrents à l'intérieur d'un même domaine. Un domaine est potentiellement réparti et peut être représenté sur plusieurs sites. Par conséquent, les activités d'un domaine peuvent s'exécuter en parallèle sur des machines différentes.

  Dans ce modèle, le partage d'objets (associé à des contraintes de synchronisation) est l'unique moyen de communication entre des activités du même domaine ou de domaines différents.
- Des objets passifs et persistants liés dynamiquement dans les domaines.

  Le modèle proposé par Guide est un modèle à objets passifs. Ses objets sont persistants et ils sont rendus accessibles aux domaines par une opération appelée *liaison*(2). Pour qu'une activité puisse appeler une méthode sur un objet, cet objet doit être lié dans son domaine.

Pour la réalisation de Guide−1, le système Unix a été choisi comme plate−forme de développement, afin de limiter la phase de développement et d'accroître la portabilité du système.

III.2 Gestion de la mémoire de stockage

Nous présentons maintenant les différentes approches pour la gestion de la mémoire de stockage.

Comme dans un système de gestion de fichiers, le nom que l'on attribue à un objet peut être relatif ou absolu, ce qui revient à gérer des contextes locaux ou globaux (section II.2.2). La mémoire de stockage étant composée d'un ensemble de partitions, un nom relatif signifie une désignation à l'intérieur d'une partition. Un nom absolu est un nom pouvant désigner tout objet quel que soit sa partition de résidence.

Ces deux approches s'expliquent par le fait qu'un nom absolu pouvant désigner tous les objets du système est naturellement plus grand (en nombre de bits) qu'un nom relatif. Pour des applications du style des bases de données où les objets se référencent souvent entre eux et où la taille de la base de données sur disque est un critère important, le fait de réduire la taille des identificateurs d'objets est appréciable. De plus, réduire la taille des identificateurs d'objets peut être également avantageux pour la vitesse d'exécution : la taille actuellement

---

(2) La liaison dans un domaine correspond à la notion de couplage dans un espace d'adressage.

utilisée (et nécessaire) pour des noms absolus d'objets dans des gestionnaires d'objets persistants est de l'ordre de 64 bits ou même 128 bits, ce qui signifie qu'un microprocesseur 32 bits sera pénalisé dans la manipulation de telles références. L'utilisation de noms relatifs permet de redescendre à une taille de 32 bits pour les noms, ce qui accélère les opérations d'affectation ou de comparaison de noms.

III.2.1 Noms relatifs

Dans le cas d'un nommage relatif, un objet ne peut contenir une référence que vers un objet de la même partition. Cela revient à avoir un contexte de résolution de noms local à la partition.

Pour permettre aux objets de faire référence à des objets appartenant à d'autres partitions, le système doit gérer des objets spéciaux qui sont des *liens de poursuite* vers des objets externes à la partition. Un tel objet contient l'identification de la partition contenant cet objet, ainsi que l'identification de l'objet dans cette partition. La localisation d'un objet est faite tout d'abord dans la partition contenant l'objet contenant la référence utilisée. Le système détecte alors si l'objet référencé dans la partition courante est ou non un lien de poursuite. Si c'est le cas, le lien de poursuite est utilisé pour atteindre l'objet dans sa partition de résidence.

Avec des noms relatifs, la localisation à l'intérieur d'une partition est un problème simple, car le nombre d'objets est en général faible. La désignation locale à une partition peut par exemple contenir l'indice de l'objet désigné dans une table des objets de la partition. De plus, il y indépendance du choix des noms d'objets entre les partitions, puisque ces noms sont internes aux partitions.

La migration entre partitions est réalisée en recopiant l'objet dans la partition cible et en remplaçant l'ancienne copie par un lien de poursuite. Ainsi, tous les noms d'objets répandus dans le système restent valides. On peut raccourcir lors des accès les chaînes de liens de poursuite lorsqu'elles sont de longueur supérieure à un lien.

Avec les liens de poursuite se pose toutefois un problème d'identité, car un nom local ne suffit pas pour désigner un objet de façon unique :

- Le même nom peut désigner des objets différents dans deux partitions différentes.

  Avec ce nommage relatif, on ne peut pas rendre accessibles en mémoire d'exécution les noms d'objets de la même manière qu'en mémoire de stockage, car deux objets différents dans deux partitions différentes peuvent avoir le même nom. Ce problème est généralement résolu au niveau de la mémoire d'exécution en remplaçant le nom relatif en mémoire de stockage par une autre désignation qui discrimine les objets.

- Deux noms différents (dans la même ou différentes partitions) peuvent désigner le même objet.

  Il faut notamment s'en rendre compte lorsque l'on charge un objet en mémoire d'exécution pour ne pas le faire plusieurs fois. Lorsqu'un objet doit être localisé, son nom est résolu dans la partition de l'objet contenant ce nom et des liens de poursuite sont suivis si nécessaire. On peut alors identifier un objet de façon unique par le nom de sa partition de résidence et son nom local dans cette partition. Une autre

technique consiste à allouer un nom unique dans tout le système pour chaque objet et à stocker ce nom dans l'état de l'objet.

On trouve des réalisations de nommages relatifs dans les projets Mneme [Moss90] et Thor (section III.1.2).

Dans le projet Thor, un nom d'objet (*Oref*) est local à un *OR* (*Object Repository*) et sa taille est de 32 bits sur les architectures actuelles (la taille d'une adresse virtuelle). Les liens de poursuite (*forwarders*) contiennent une référence globale à tout le système (*Xref*) composée de l'identification de l'*OR* de résidence (supposée) de l'objet et de l'*Oref* de l'objet dans cet *OR*.

Dans Thor, les *Orefs* contenus dans un objet sont transformés en adresses virtuelles lorsque l'objet est amené en mémoire d'exécution (cette technique appelée *mutation de pointeur* ou *swizzling* est décrite en détail dans la section III.3). C'est à ce moment (chargement) que le système distingue les objets ayant le même nom local en tenant compte de l'*OR* de provenance de l'objet. Ainsi, les *Orefs* ne sont jamais rendus visibles aux programmes clients qui ne voient que des adresses virtuelles, ce qui résout le problème de la localité des noms. Pour le problème dû au fait que différents *Orefs* peuvent faire référence au même objet, Thor stocke un identifiant unique (*Uid*) dans chaque objet.

En résumé, les principales caractéristiques des mémoires de stockage à nommage relatif sont les suivantes :

- Noms d'objets petits et relatifs à la partition contenant le nom,
- Références inter−partitions par lien de poursuite,
- Migration par lien de poursuite,
- Gestion nécessaire de la distinction des noms en mémoire d'exécution.

III.2.2 Noms absolus

On oppose au nommage relatif le nommage absolu, avec lequel un objet peut désigner directement tout objet du système. Cela correspond à avoir un contexte global de résolution de noms. Dans ce cas, la localisation d'un objet peut être plus ou moins aisée en fonction de la façon dont son nom lui est attribué. Ce nom, qui doit désigner l'objet de façon unique, peut contenir des informations sur sa localisation afin de permettre de le retrouver plus rapidement ; on dit alors que le nom est dépendant de la localisation de l'objet.

Noms absolus indépendants de la localisation

Si les noms des objets sont indépendants de leur localisation, alors la localisation est généralement assurée par un ensemble de serveurs coopérants répartis sur le réseau de stations de travail. A chaque fois qu'un objet doit être localisé sur disque, cette organisation de serveurs doit être interrogée afin de déterminer dans quelle partition se trouve cet objet, ce qui est pénalisant pour les performances. Par contre, cette solution a l'avantage de simplifier la gestion de la migration des objets, car la migration d'un objet doit seulement être prise en compte au niveau des serveurs de localisation pour être totalement réalisée et elle ne modifie pas le schéma de localisation.

On trouve une illustration de cette technique dans le projet Clouds (section III.1.3). Les noms d'objets de Clouds (*Sysnames*) sont indépendants de la localisation de l'objet. Ils sont stockés dans des *Partitions* qui sont des serveurs coopérant à la gestion des segments répartis sur le réseau. Chaque serveur gère un ensemble de segments sur disque et participe à la tâche de localisation des segments du système.

Différentes stratégies de gestion de ces serveurs coopérants sont présentées dans [Legatheaux88].

Noms absolus dépendants de la localisation

Supposons maintenant que des informations de localisation soient incluses dans le nom de chaque objet afin d'accélérer leur localisation. Dans ce cas, l'attribution du nom de l'objet lors de sa création est fonction de l'endroit où l'objet est créé et la localisation de l'objet est directe, car l'identification de la partition contenant l'objet est insérée dans le nom de l'objet. Le problème d'une telle solution est la réalisation de la migration d'objet. Nous passons en revue les différentes améliorations en partant de la solution fondée sur les liens de poursuite :

- Les liens de poursuite.

  Comme dans le cas des noms relatifs, on laisse des liens de poursuite dans les partitions pour chaque objet déplacé. Ainsi, la demande d'accès à un objet est systématiquement envoyée à la partition de création, puis éventuellement transmise de lien de poursuite en lien de poursuite jusqu'à ce que l'objet soit trouvé.

  Cette solution a l'inconvénient de pénaliser toutes les migrations sans ne jamais y remédier. On peut raccourcir les chaînes de liens de poursuite lorsqu'elles sont de longueur supérieure à un lien, mais on ne peut pas obtenir le coût de localisation initial pour un objet déplacé.

- Le changement de nom.

  Cette solution consiste à ajouter à la précédente un changement de nom pour les objets qui migrent. Lorsqu'un nom d'objet est stocké dans un objet et lorsque l'utilisation de ce nom montre que l'objet désigné a été déplacé, le nouveau nom de l'objet, incluant les nouvelles informations sur sa localisation, est affecté dans l'objet contenant ce nom. Ainsi, les futures utilisations de ce nom bénéficieront d'une localisation directe.

  Le problème posé par une telle solution est qu'un même objet peut être désigné par plusieurs noms (comme dans le cas des noms relatifs). Il faut alors être capable de comparer deux noms d'objets, afin de savoir s'il s'agit du même objet. Ceci signifie que la comparaison de deux références à des objets implique une résolution de ces références, pour arriver aux localisations finales des objets et déterminer s'il s'agit du même objet, ce qui est très pénalisant.

- Ajouter dans les identificateurs d'objets des informations sur la localisation courante.

  Le nom d'un objet peut être composé de deux champs *identité* et *localisation.* Cette technique, qui est une variante de la précédente, consiste à augmenter la

taille du nom d'un objet pour qu'il puisse contenir le nom qui lui est attribué à la création de l'objet et qui sert d'identificateur unique, ainsi que des informations sur sa localisation courante. La première partie sert à identifier l'objet sans qu'un accès à cet objet soit nécessaire. La seconde partie du nom sert à localiser l'objet et peut être modifiée à chaque action de localisation, ce qui lui permet d'indiquer la dernière localisation connue. Des liens de poursuite sont toujours gérés pour les localisations dont la deuxième partie n'est pas à jour.

On évite ainsi de localiser des objets pour comparer leur identité, mais la taille du nom des objets peut devenir une contrainte très gênante.

Cette solution est utilisée dans le système Guide−1 (section III.1.9) dans lequel un appel de méthode peut modifier une partie de la référence utilisée si l'objet a été déplacé.

- Un ensemble de serveurs de migration.

  Cette solution revient à utiliser un ensemble de serveurs répartis coopérants qui enregistrent les déplacements des objets.

  La localisation d'un objet commence par utiliser les informations présentes dans son nom ; si l'objet n'est pas trouvé dans la partition indiquée (la partition de création), un serveur de migration est appelé pour déterminer la partition contenant l'objet. L'avantage de cette solution, par rapport à la solution fondée sur des serveurs et où le nom est indépendant de la localisation, est d'optimiser l'accès aux objets qui n'ont pas été déplacés : le serveur de migration n'est contacté que pour les objets déplacés, le nombre d'objets concernés devant rester faible. Son avantage par rapport aux solutions fondées sur des liens de poursuite est d'éviter le parcours d'une chaîne de liens de poursuite et de faire payer le même coût quel que soit le nombre de migrations d'un objet.

  Une autre variante consiste à contacter en premier un service local qui enregistre les localisations déjà effectuées (un cache). Si l'objet recherché n'est pas dans ce cache, la méthode précédente est utilisée. On accélère ainsi la localisation des objets déplacés souvent utilisés.

  Cette technique est utilisée dans le système Gothic (section III.1.8). Dans Gothic, un cache géré sur chaque site mémorise les localisations des objets déjà référencés. Si la localisation de l'objet n'est pas dans le cache, un message de localisation est envoyé au site de création de l'objet (donné par son nom). Si l'objet est dans le cache, le message de localisation est envoyé au site donné par le cache. Le site recevant un message de localisation retourne l'objet. Si l'objet a été déplacé depuis, une diffusion de message à tous les sites est utilisée.

En résumé, les possibilités de mémoires de stockage à base de noms absolus sont donc les suivantes :

- Noms indépendants de la localisation

  La localisation et la migration sont fondées sur la gestion de serveurs enregistrant la localisation des objets dans le système.

- Noms dépendants de la localisation

  La localisation d'un objet qui n'a pas été déplacé est directe grâce aux informations dans son nom. La migration peut être gérée par serveurs de migration ou liens de poursuite.

III.2.3 Conclusion

Nous avons étudié les grandes approches possibles pour la gestion de la mémoire de stockage qui sont :

- Le nommage relatif, qui est principalement motivé par une réduction de la taille des identificateurs d'objets, mais qui reporte des problèmes au niveau de la mémoire d'exécution comme celui de la comparaison de noms.
- Le nommage absolu, qui peut être dépendant ou indépendant de la localisation. L'avantage d'opter pour un nommage des objets indépendant de la localisation des objets est de faciliter la migration des objets, puisque le processus de localisation n'est pas affecté par la migration de l'objet. L'avantage d'inclure dans un nom des informations sur la localisation de l'objet est de rendre plus efficace cette localisation, mais c'est généralement au détriment de la réalisation de la migration des objets.

La section suivante est consacrée aux méthodes de gestion de la mémoire d'exécution.

III.3 Gestion de la mémoire d'exécution

La mémoire d'exécution peut être organisée de nombreuses façons. Les deux abstractions essentielles gérées dans cette mémoire sont l'objet et l'activité, qui est un flot d'exécution séquentiel.

Une des motivations pour cette organisation est la protection (prenant ici le sens de confinement d'adressage). Il est en effet nécessaire de confiner un objet ou une activité dans un espace afin de l'isoler d'une part des perturbations possibles provenant d'une erreur de comportement d'un autre (objet ou activité) et d'autre part pour l'empêcher de nuire aux autres. L'outil de base pour garantir ce confinement est le domaine de protection. Nous allons donc étudier dans la section III.3.2 comment les domaines de protection peuvent être utilisés pour l'organisation de cette mémoire d'exécution.

Une fois cette organisation décrite, nous étudierons comment le partage d'objets peut être réalisé. Il faut permettre le partage d'objets entre des activités pouvant s'exécuter dans des domaines de protection différents, pouvant être sur des machines différentes. La section III.3.3 présente les différentes techniques.

Enfin, lorsqu'un objet doit être adressé à partir de son nom, deux phases doivent être mises en œuvre, la première permettant de rallier un domaine de protection dans lequel on peut l'adresser, en fonction des règles choisies pour le partage et la protection, et la deuxième permettant de récupérer l'adresse de l'objet dans l'espace virtuel dans lequel il est couplé. L'adressage des objets est abordé dans la section III.3.4.

La section III.3.5 conclut par une synthèse des différentes mémoires d'exécutions que l'on rencontre dans les projets de recherche.

Mais avant de développer ces parties, nous allons définir plus précisément les termes que nous utiliserons et avons déjà utilisés, à savoir : **espace virtuel**, **domaine de protection**, **couplage dans un espace virtuel** et **couplage dans un domaine de protection** (section III.3.1).

III.3.1 Définitions

Espace virtuel

Nous appelons Espace Virtuel l'ensemble des adresses interprétables par un processeur. Les stations de travail utilisées sont pour la plupart pourvues de MMU (Memory Management Unit) permettant la gestion de mémoire virtuelle. Cette gestion de la mémoire virtuelle résout de nombreux problèmes :

- Un espace virtuel est de taille bien plus grande que la taille de la mémoire centrale. La mémoire virtuelle libère l'utilisateur de la gestion de zones de recouvrement ; le remplacement de pages en mémoire centrale est géré par le mécanisme de pagination.
- Elle libère l'utilisateur de la gestion des traductions d'adresse ; l'utilisateur (un compilateur) utilise des adresses virtuelles dans un espace fictif et le processeur se charge des traductions entre les adresses virtuelles et les adresses réelles en mémoire centrale.

Un espace virtuel peut également être vu comme un contexte de résolution de nom(3) dans lequel les noms sont des adresses virtuelles. La liaison est l'opération permettant d'associer un nom à un objet. Cette opération de liaison dans ce contexte de résolution de nom est généralement appelée **couplage de l'objet dans l'espace virtuel**.

A un instant donné, une activité s'exécute dans un seul espace virtuel. Lorsqu'un objet est couplé à une adresse donnée dans un espace virtuel, cela signifie que l'utilisation de cette adresse par une activité dans cet espace virtuel permet de désigner cet objet si l'objet est accessible par cette activité. L'accessibilité d'un objet par une activité dépend de la notion de domaine de protection.

Domaine de protection

Une capacité est un moyen d'accéder à un objet. Elle contient une adresse et un ensemble de droits d'accès à l'objet. Un domaine de protection(4) est un ensemble de capacités sur des objets. Une activité s'exécute dans un seul domaine de protection à un instant donné, elle ne peut accéder qu'aux objets pour lesquels une capacité figure dans le domaine.

---

(3) Un contexte de résolution de nom, définit par J.H. Saltzer, a été décrit au chapitre II.

(4) Nous utiliserons le terme domaine pour domaine de protection.

Une capacité spéciale est en général associée à chaque domaine de protection. Une activité a la possibilité de changer de domaine de protection en utilisant cette capacité.

On appelle **couplage dans un domaine de protection** l'opération qui consiste à rendre un objet accessible dans un domaine, en y ajoutant une capacité.

Une activité s'exécute à un instant donné dans un unique domaine et dans un unique espace virtuel, mais plusieurs activités peuvent en général partager un domaine ou un espace virtuel.

Les domaines de protection sont indépendants des espaces virtuels gérés. Une activité peut changer de domaine sans changer d'espace virtuel et vice−versa. Il est possible de gérer un espace virtuel global à tous les domaines, ou de gérer un espace virtuel propre à chaque domaine.

La notion de processus Unix correspond à une activité s'exécutant dans un domaine de protection qui a un espace virtuel privé.

III.3.2 Organisation de la mémoire d'exécution

La mémoire d'exécution peut être organisée à partir de la notion de domaine de protection.

Pour étudier les différentes possibilités pour réaliser une mémoire d'exécution, nous allons partir d'une version minimale, puis étudier l'influence des fonctions à assurer (distribution et protection) sur la réalisation de cette mémoire.

La version minimale est obtenue en supprimant les problèmes liés à la distribution et à la protection. Dans ce cas, il suffit d'utiliser un seul domaine sur le site concerné. Les objets sont alors couplés à la demande dans ce domaine et toutes les activités de toutes les applications s'exécutent dans ce domaine.

III.3.2.1 Distribution

Si l'on ajoute le problème de la distribution, il devient nécessaire de gérer au moins un domaine par site si on veut profiter des possibilités de parallélisme offertes par la présence de plusieurs machines (un domaine étant local à une machine). Notre version minimale devient alors une version dans laquelle toutes les activités et tous les objets sur le même site s'exécutent dans le même domaine, avec un domaine par site.

C'est le cas notamment dans le projet Emerald (section III.1.5) réalisé sur le système Unix. Un unique processus Unix sur chaque site contient le noyau d'Emerald ainsi que tous les objets en cours d'exécution sur ce site. L'ordonnancement des activités dans ce processus Unix est réalisé par le noyau d'Emerald et les activités sur un site partagent toutes le domaine de protection de ce processus. De même, le système Gothic (section III.1.8) réalisé sur le micro−noyau Mach 3.0 couple tout objet utilisé sur un site dans la même tâche Mach[(5)]. Le domaine de protection associé à cette tâche Mach est partagé par tous les flots d'exécution activés dans la tâche.

---

(5) A ce stade, une tâche Mach peut être vue comme un processus Unix.

Dans ces deux cas, aucune protection physique entre les objets n'est fournie. La protection des objets ne peut être assurée que par les langages de programmation en supposant que toutes les applications sont écrites avec des langages sûrs(6).

III.3.2.2 Protection

A l'exception des contrôles effectués par les langages de programmation, la seule protection réelle est celle fournie par la séparation entre les différents domaines de protection. Cette séparation des domaines peut être utilisée pour créer des cloisonnements entre les activités et entre les objets.

Séparation des activités

Dans un système, une erreur dans l'exécution d'une application s'exécutant pour le compte d'un utilisateur ne doit pas perturber l'exécution d'autres applications. Cette propriété peut être plus ou moins respectée en fonction de l'organisation des structures d'exécution en mémoire d'exécution.

Ainsi, il ne faut pas que deux activités qui s'exécutent pour le compte d'applications différentes puissent se perturber. Une telle perturbation, si elle provient du partage d'objets entre les applications, ne peut pas être évitée. Mais il ne faut pas que cette perturbation vienne d'une source différente. Dans la version précédente où tous les objets et toutes les activités en mémoire d'exécution sur un site sont dans le même domaine de protection, une activité peut adresser par erreur un objet qu'elle ne partage pas (dans le cadre de son application), donc perturber une activité avec laquelle elle n'a rien en commun. L'objectif de la séparation des activités est donc de limiter les facultés d'adressage d'une activité au strict nécessaire. Cette propriété est généralement obtenue en isolant chaque activité dans un domaine de protection et en n'y couplant que les objets qu'elle partage.

C'est le cas dans le projet Thor (section III.1.2). Dans Thor, un processus (*FE*) voulant utiliser un objet le charge dans le domaine de protection qui lui est associé. Ainsi, ce processus ne peut adresser que les objets effectivement chargés dans ce domaine.

Par contre, dans le projet Guide−1 (section III.1.9), bien que chaque activité de Guide soit réalisée par un processus Unix et se voit donc associer un domaine de protection privé, l'isolation des activité n'est pas assurée. En effet, tout objet présent en mémoire d'exécution sur un site est chargé dans un grand segment de mémoire partagé par tous les processus du site. En conséquence, dès qu'un objet est utilisé par une activité, il est accessible par toutes les autres.

L'isolation entre les activités procure un certain degré de protection, mais il est possible d'assurer un meilleur cloisonnement en isolant les objets entre eux.

Séparation des objets

En théorie, la règle d'encapsulation des objets voudrait qu'un seul objet soit adressable à un instant donné par une activité. Ainsi, une erreur d'adressage dans l'exécution d'une méthode ne pourrait atteindre que les données de l'objet dont une méthode s'exécute(7).

---

(6) La sûreté d'un langage se traduit par une confiance dans le code généré par le compilateur.

(7) Si ce degré de protection est assuré, l'isolation des activités l'est également.

Cette propriété peut être obtenue en faisant en sorte qu'il n'y ait qu'un seul objet par domaine, soit statiquement, soit dynamiquement :

- Statiquement.

  A sa création, un domaine se voit attribuer un seul et unique objet. Ainsi, chaque objet se trouve isolé des autres objets et la seule façon d'adresser un objet est alors d'appeler une méthode de cet objet, en changeant par conséquent de domaine.

  Cette technique est notamment celle choisie dans le projet Argus (section III.1.1) où les objets sont des objets actifs réalisés en mémoire d'exécution par des processus sur le système Unix. Chaque objet est dans un domaine privé et l'appel de méthode entre objets est réalisé par envoi de messages entre les processus contenant l'objet appelant et l'objet appelé. Un processus Unix est alors un serveur pour un unique objet.

  Le problème d'une telle approche est la taille des objets : un objet étant un processus Unix, on ne peut réaliser de la sorte un petit objet sans gaspiller les ressources du système. De plus, chaque appel à un objet entraîne un échange de message. Pour éviter le coût inacceptable d'une commutation de domaine pour chaque appel, le système Argus gère au niveau langage des objets de plus petite taille. Ces objets appelés objets locaux ne sont pas visibles de l'extérieur de l'objet global les contenant. Un appel entre objets locaux est réalisé par appel de procédure.

- Dynamiquement

  Un seul objet est présent dans un domaine à un instant donné et il est possible de changer l'objet accessible dans un domaine. Cette technique est utilisée dans le projet Clouds (section III.1.3). Dans Clouds, un objet n'est couplé dans un domaine que pendant le temps d'exécution d'une méthode de cet objet. Chaque appel de méthode provoque une annulation du couplage de l'objet appelant et le couplage de l'objet appelé. Les objets sont alors isolés, puisque seul l'objet appelé est adressable pendant un appel de méthode.

  Les inconvénients de cette façon d'opérer sont également la taille des objets et le coût de l'appel de méthode : dans Clouds, les objets sont gros et chaque appel de méthode nécessite un appel au noyau pour coupler l'objet appelé.

Dans les deux cas, la taille des objets comparée au coût de gestion des domaines de protection incite donc au regroupement d'objets et à assurer le confinement pour un ensemble d'objets. Différents critères de regroupement d'objets peuvent être utilisés et offrent différents types de protection. Nous en citons quelques uns.

- Un exemple est le regroupement par propriétaire d'objet. On n'autorise le couplage de plusieurs objets dans le même domaine que s'ils appartiennent au même propriétaire. Ainsi, la protection ne peut être violée que par un objet du même utilisateur.
- Un autre exemple est le regroupement des objets utilisés dans le cadre de la même application. Ainsi, seuls les objets explicitement utilisés par l'application peuvent être adressés.

Nous verrons dans les chapitres suivants comment nous assurons ces deux types d'isolation dans le cadre du projet Guide−2.

En résumé, la mémoire d'exécution est composée d'un ensemble de domaines de protection et d'activités s'exécutant dans ces domaines. La notion de domaine est principalement utilisée pour deux raisons :

- La répartition des objets et des activités en mémoire d'exécution passe par la gestion de plusieurs domaines répartis sur le réseau de machines.
- L'isolation des objets et des activités nécessite que ces entités soient confinées dans des domaines de protection différents.

III.3.3 Partage

On se pose maintenant le problème du partage d'objets entre activités en mémoire d'exécution.

Dans le partage d'objets entre activités, il faut distinguer le partage concurrent où des activités partagent l'objet au même instant, du partage sérialisé ou l'objet est partagé par des activités sur des périodes ne se recouvrant pas.

Lorsque le partage est concurrent entre différentes activités, il faut distinguer les cas où :

- Les activités sont dans le même domaine.
- Les activités sont dans des domaines différents sur la même machine.
- Les activités sont dans des domaines différents sur des machines différentes.

III.3.3.1 Partage sérialisé

Dans le cas du partage sérialisé d'un objet, une seule activité est autorisée à utiliser cet objet à un instant donné. Ainsi, il est possible de délivrer une copie de l'objet à des activités différentes sur la même machine ou sur des machines différentes quels que soient leurs domaines de protection d'exécution, car ces copies ne seront jamais données sur des périodes se recouvrant. La sérialisation doit être assurée en utilisant une technique de verrouillage des objets.

Cette technique est notamment utilisée dans des systèmes transactionnels de gestion de base de données comme dans les projets Mneme [Moss90] et Thor (section III.1.2). Dans Thor, le verrouillage des objets par les transactions implique qu'un objet ne peut être chargé que dans un processus (*FE*) à la fois. Ce verrouillage est d'autant plus nécessaire que, chaque processus ayant un espace virtuel privé et les noms d'objets étant remplacés par des adresses virtuelles en mémoire d'exécution (*swizzling*), il serait difficile d'assurer que ces adresses soient cohérentes dans tous les processus partageant les objets (ce problème est traité plus en détail en section III.3.4).

III.3.3.2 Partage dans le même domaine

Le partage d'objets entre activités dans le même domaine est implicite.

III.3.3.3 Partage entre domaines différents sur la même machine

Lorsque deux activités dans deux domaines différents doivent partager un objet, il est toujours possible de se ramener au cas précédent du partage dans le même domaine en utilisant l'opération de changement de domaine. Un exemple simple de cette technique est l'appel de procédure à distance entre ces deux domaines.

Mais pour des raisons d'isolation des applications entre elles (confinement), le modèle d'exécution du système peut imposer le partage d'objets entre des activités dans des domaines différents. Le mécanisme de couplage permettant le partage simultané de mémoire entre domaines différents existe dans la plupart des systèmes. Il se distingue de l'opération de chargement par le fait que si plusieurs domaines couplent le même objet pour le partager, ils travailleront sur le même objet et non sur des copies différentes.

Sur une même machine, le partage de mémoire entre domaines n'est pas un concept nouveau et il est fourni en mettant en correspondance des morceaux de tables des pages dans le noyau. Un exemple de partage de mémoire entre domaines de protection est le partage de mémoire entre processus sur le système Unix.

III.3.3.4 Partage entre domaines différents sur des machines différentes

Partage par migration

Ici aussi, il est possible de se ramener au cas précédent, en utilisant un mécanisme permettant de réaliser le partage effectif sur la même machine.

Le principe du partage par migration est de n'autoriser le partage que sur un seul site à la fois. Cette exclusion mutuelle entre les sites candidats à l'accès à l'objet est compensée par des mécanismes garantissant la disponibilité des objets :

- La migration des activités

  Une activité voulant accéder à un objet déjà couplé sur un autre site change de site d'exécution pour aller sur le site où est l'objet, le partage se faisant donc de façon locale. Une fois les deux activités sur le même site, le partage en local peut être réalisé avec les deux techniques données précédemment, à savoir en partageant dans le même domaine ou par couplage dans des domaines différents sur la même machine.

  Les systèmes Guide−1 (section III.1.9) et Amber (section III.1.6) utilisent cette technique. Le système Guide−1 a été réalisé sur Unix. Dans Guide−1, une zone de mémoire partagée est allouée sur chaque site et couplée par tous les processus Unix du site. Chaque activité est réalisée par un processus Unix. Un objet en mémoire d'exécution ne peut être chargé en mémoire partagée que sur un site. Si une activité sur un site veut appeler un objet déjà chargé sur un autre site, un mécanisme d'appel à distance est utilisé, qui transfert l'exécution de l'activité sur le site où l'objet est chargé.

- La migration des objets

  Une activité voulant accéder à un objet déjà couplé sur un autre site demande le découplage de l'objet sur ce site et le couplage sur son site. Le partage se fera toujours de façon locale, mais c'est cette fois−ci l'objet qui va à l'activité. Le

problème d'une telle solution est la gestion des activités en cours d'exécution sur cet objet. Elles sont généralement déplacées avec l'objet.

Le système Amber (section III.1.6) permet également la migration des objets. Un objet qui est couplé sur une autre machine est par défaut accédé par migration de l'activité, mais il est possible de changer le site de couplage sur requête explicite de l'utilisateur.

C'est en pratique un compromis entre ces deux possibilités qui est employé en fonction du taux d'utilisation des objets sur les sites : si l'objet est très utilisé sur un site, la migration d'activité sera préférée à la migration d'objet. Les techniques de migration d'objets et d'activités permettent également de réguler la charge sur les stations de travail du réseau.

Partage par couplage et copies multiples

Pour le partage entre machines différentes, une autre technique consiste à étendre le mécanisme de couplage pour permettre la gestion de copies multiples sur les sites partageant l'objet. Le principal problème est alors de garantir la cohérence des données partagées, un objet pouvant être adressé simultanément sur des machines différentes. Différents niveaux de cohérence peuvent être offerts [Banâtre91]. La cohérence la plus couramment réalisée est la cohérence du type lecteur/rédacteur strict : des lectures peuvent être effectuées simultanément sur plusieurs machines, mais toutes les écritures doivent être vues dans le même ordre sur toutes les machines. Il est toutefois possible d'offrir des cohérences moins restrictives [Boyer91b].

Il existe différentes techniques permettant le couplage entre machines différentes et garantissant cette cohérence. Cette cohérence peut être assurée par va−et−vient, ou par diffusion :

- Cohérence par va−et−vient

  Le partage avec cohérence par va−et−vient est caractérisé par : un protocole qui détermine les conditions dans lesquelles des copies peuvent être données aux différentes machines et l'opération de préemption de copie qui permet de reprendre une copie donnée à un site, afin de rendre à nouveau cette copie disponible pour un autre site candidat au partage.

  L'unité de partage, de copie et de préemption peut être l'objet ou la page, ce qui signifie que la détection du type de l'accès (lecture ou écriture) peut être faite au niveau de l'objet ou de la page. Une tentative d'accès à un objet (resp. page) non présent sur le site provoque un défaut d'objet (resp. page). Un défaut est traité en donnant une copie de l'objet (resp. page) au site demandeur. Ce don peut nécessiter auparavant une préemption sur un ou plusieurs autres sites.

  Si l'unité de partage est l'objet, il faut détecter à la compilation pour chaque méthode si l'objet sera accédé (potentiellement) en lecture ou en écriture. L'exécution d'une méthode vérifie alors la présence de l'objet dans le bon mode et provoque un défaut d'objet si ce n'est pas le cas.

  Le maintien de la cohérence au niveau de la page lui est généralement préféré [Li89], car il repose sur une détection matérielle du type de l'accès. Une page peut être délivrée à un site en mode écriture ou à plusieurs sites en mode lecture. Une

tentative d'accès à une page dans un mode non autorisé provoque un défaut de page, pouvant provoquer la préemption de cette page sur un autre site, pour traiter ce défaut. Cette technique ne peut être réalisée qu'à un niveau très proche de la machine (dans le gestionnaire de pagination). L'arrivée des micro−noyaux permettant de réaliser le contrôle de la pagination dans un serveur externe au noyau a généralisé ce type d'approche.

On retrouve cette approche dans le projet Clouds (section III.1.3). Les objets sont de grosse taille et composés d'un ensemble de pages. Un objet couplé sur un site est désigné par une table des pages sur ce site. Lorsqu'elle appelle un objet, une activité, qui s'exécute dans un domaine privé, commute un registre de table des pages, ce qui revient à coupler l'objet dans son domaine. Un objet peut être couplé simultanément par des activités sur plusieurs sites. Les objets sont stockés dans des *Partitions* qui sont des serveurs de stockage. Ces serveurs répondent aux défauts de page sur les objets et maintiennent la cohérence de ces données suivant un protocole lecteur/rédacteur.

Cette technique pour le partage des objets est également utilisée dans les projet Opal (section III.1.7) et Gothic (section III.1.8).

- Cohérence par diffusion

  En cohérence par diffusion, des copies des données partagées résident sur toutes les machines les partageant et les modifications de ces données sont diffusées à tous les sites propriétaires d'une copie.

  Cette diffusion doit garantir que les modifications sont faites dans le même ordre sur toutes les machines. Pour ce faire, un séquenceur doit être utilisé pour ordonner les messages. On distingue ici deux techniques correspondant à deux unités de maintien de la cohérence. La première consiste à choisir comme unité l'objet et à diffuser les appels de méthodes qui modifient les objets, pour réexécuter ces appels de méthodes sur les copies existantes. La seconde consiste à choisir le mot mémoire comme unité et à diffuser les nouvelles valeurs pour les affecter sur les copies existantes.

  La cohérence par diffusion est avantageuse lorsqu'il y a un faible taux d'écriture, les lectures pouvant se faire en parallèle sur les sites partageant les objets.

  Elle est utilisé dans le projet Orca (section III.1.4). Les opérations se voient attribuer lors de la compilation un type qui est "lecture seule" ou "modification possible". Une opération de type "lecture seule" utilise directement sa copie locale, alors qu'une opération de type "modification possible" diffuse les paramètres de l'opération aux sites ayant une copie pour réexécution sur ce site. L'inconvénient de cette solution est qu'une analyse statique des programmes risque d'attribuer un type "modification possible" à toutes les opérations (ou du moins la majorité).

En résumé, au niveau de la mémoire d'exécution, le partage d'objets entre activités pouvant s'exécuter dans des domaines différents sur des machines différentes peut être réalisé :

- par couplage. Le couplage permet à l'activité d'exécuter les appels de méthode localement dans le domaine courant. Le système doit alors garantir la cohérence des données lorsqu'elles sont partagées entre plusieurs machines.
- par migration. La migration permet de se ramener à un cas de partage local sur la même machine, ou même de partage dans le même domaine.

III.3.4 Adressage des objets

Nous allons maintenant étudier les principes de réalisation de la résolution des noms d'objets en mémoire d'exécution.

A chaque appel de méthode sur un objet, il faut être en mesure d'adresser les données de l'objet. Il faut donc que l'activité courante s'exécute dans un domaine de protection autorisé à coupler l'objet et que l'objet appelé soit couplé dans ce domaine. Puis il faut que l'objet soit couplé dans un espace virtuel pour être adressable par une adresse virtuelle et obtenir à partir du nom de l'objet son adresse virtuelle de couplage.

Une première phase consiste donc à faire en sorte que l'objet soit accessible, c'est à dire à réunir l'objet et l'activité dans le même domaine de protection. Cette phase peut se traduire :

- par le couplage de l'objet appelé dans le domaine d'exécution de l'activité,
- ou par une migration de l'activité vers un autre domaine, pour permettre le partage, pour des raisons de protection, ou pour les deux raisons.

Pour cette première phase, il est nécessaire de gérer un contexte global de résolution de noms indiquant soit le couplage nécessaire, soit le domaine de protection à rallier pour que l'objet soit accessible. Cette résolution de nom est généralement interprétée (il s'agit d'une gestion de table de traduction entre le nom de l'objet et l'information recherchée). Le coût de cette première phase est lié à la technique utilisée pour réaliser le partage et au degré de protection fourni. Elle aura lieu à chaque appel d'objet si chaque objet est confiné dans un domaine privé. Cette phase est appelée *défaut d'objet*.

Dans une seconde phase, l'objet étant accessible dans le domaine courant, le problème est d'obtenir une adresse virtuelle à partir du nom de l'objet. Il est alors nécessaire de gérer un ou plusieurs contextes de résolution de noms associant une adresse virtuelle à un nom d'objet, ce qui revient à gérer un ou plusieurs espaces virtuels. Les deux critères permettant de distinguer les cas envisageables sont les suivants :

- le couplage des objets dans les espaces virtuels (et la liaison dans les contextes) est réalisé statiquement (à la création de l'objet) ou dynamiquement (lors de l'exécution),
- on gère un ou plusieurs espaces virtuels (ce qui implique la gestion de un ou plusieurs contextes de résolution de nom).

Et les cas envisageables sont alors les suivants :

- Couplage dynamique dans plusieurs espaces virtuels,
- Couplage dynamique dans un espace virtuel unique,

- Couplage statique dans plusieurs espaces virtuels,
- Couplage statique dans un espace virtuel unique.

Nous les passons en revue.

III.3.4.1 Couplage dynamique dans plusieurs espaces virtuels

Les objets sont couplés dynamiquement (à l'exécution) dans différents espaces virtuels et peuvent donc être couplés à des adresses virtuelles différentes (sinon, on a un espace virtuel unique). Un contexte de résolution de noms est associé à chaque espace virtuel et donne pour chaque nom d'objet son adresse de couplage dans cet espace virtuel. La gestion du contexte pour la résolution des noms prend généralement la forme d'une table dont la fonction d'accès est fonction du nom de l'objet (généralement un adressage dispersé).

Des raccourcis d'adressage sont en général employés, afin d'éviter une résolution de nom systématique (aussi appelée interprétation). On peut distinguer le cas où un objet n'est jamais couplé simultanément dans plusieurs espaces virtuels (couplage sérialisé) du cas où un objet peut être couplé dans plusieurs espaces virtuels au même instant (couplage concurrent) :

- Couplage sérialisé

  Lorsqu'un nom d'objet doit être résolu, ce nom d'objet est contenu dans une variable qui peut être une variable temporaire sur la pile, ou une variable d'état d'un objet (temporaire ou persistant). Le principe du raccourci consiste à modifier cette variable et à l'enrichir d'une information permettant de résoudre plus rapidement le nom qu'elle contient lors des prochaines résolutions. Un raccourci très répandu dans le domaine, appelé *mutation de pointeur* (*swizzling*), consiste à remplacer le nom de l'objet par son adresse virtuelle de couplage. Un exemple d'utilisation de cette technique est d'opérer cette substitution à la première utilisation. Ainsi, les utilisations suivantes de cette variable trouveront l'adresse de couplage de l'objet à la place du nom.

  Ce raccourci n'est pas utilisable dans le cas du couplage concurrent, car le pointeur remplaçant le nom de l'objet ne serait pas valide dans tous les espaces virtuels où l'objet est couplé.

  Pour utiliser la mutation de pointeur, deux problèmes doivent être résolus : il faut détecter lors de l'utilisation d'un nom s'il a déjà été changé en adresse et il faut lors du découplage d'un objet remplacer les adresses qu'il contient par les noms des objets qu'elles désignent.

  On distingue les réalisations suivantes de la mutation :

  - La mutation paresseuse (*Lazy swizzling*)

    Un nom d'objet est transformé à sa première utilisation. Une information dans ce nom permet de détecter si le nom est déjà transformé. A l'exécution, un test doit être fait avant chaque utilisation de nom.

- La mutation anticipée (*Eager swizzling*)

  Lorsqu'un objet est couplé dans l'espace virtuel (lors du premier accès), chaque nom dans cet objet est transformé en adresse. Cette adresse est l'adresse de l'objet désigné s'il est résident ; c'est l'adresse d'un objet intermédiaire sinon. L'appel à un objet intermédiaire provoque un défaut d'objet qui couple l'objet et le substitue à l'objet intermédiaire.

  Cette solution est utilisée dans Thor (section III.1.2). Dans Thor, lorsqu'un objet est couplé en mémoire d'exécution, tous les noms qu'il contient sont transformés. Pour les objets qu'il désigne et qui sont non résidents, un objet intermédiaire (appelé *surrogate*) est créé et contient le nom de l'objet. L'utilisation d'un *surrogate* provoque le couplage de l'objet et l'adresse vers l'objet intermédiaire est remplacée par l'adresse de couplage de l'objet.

La mutation de pointeur est une technique très utilisée dans le domaine des bases de données. Elle repose en général sur une sérialisation des couplages dans les domaines de protection et dans les espaces virtuels, la sérialisation étant obtenue par les verrouillages effectués par des transactions.

- Couplage concurrent

  La mutation de pointeur ne peut pas être utilisée, puisqu'un objet peut être couplé à des adresses virtuelles différentes. De même, il est nécessaire que le code exécuté soit indépendant de sa localisation dans l'espace virtuel où il est couplé.

  Il est tout de même possible de mettre en place un raccourci en stockant dans une zone locale à l'espace virtuel courant l'adresse de résolution du nom de l'objet. L'association de cette zone et de la référence à l'objet est gérée statiquement par le compilateur, ce qui permet d'accélérer fortement la résolution de nom si cette zone est à jour.

  Cette technique a été introduite dans le système Multics [Daley68], afin de permettre le partage de segments à des adresses virtuelles différentes. A chaque segment de Multics est associé, dans tout espace virtuel le partageant, une table appelée segment de liaison, et à chaque référence externe dans ce segment est associée une entrée de cette table. Cette entrée permet de mémoriser l'adresse, dans l'espace virtuel courant, de résolution du nom du segment référencé.

  Un autre exemple nous est donné par le projet E [Schuh90] dans lequel l'utilisation d'un nom d'objet ne peut se faire que par l'utilisation d'une variable locale.Une telle variable étant créée sur le pile, elle est par conséquent locale à l'espace virtuel courant et peut être remplacée (mutation de pointeur) par l'adresse de couplage de l'objet dans l'espace virtuel courant. Dans ce cas, aucune restitution de nom d'objet n'est à faire. L'inconvénient est la perte des variables commutées lorsqu'elles sont dépilées.

III.3.4.2 Couplage dynamique dans un espace virtuel unique

L'objet n'est couplé que dans un unique espace virtuel (indépendamment de la gestion des domaines de protection). Un seul contexte de résolution de noms d'objets est géré pour tout le système.

La technique de raccourci appelée mutation de pointeur décrite en III.3.4.1 est également applicable dans ce cas, puisque chaque objet ne peut se voir associer qu'une seule adresse virtuelle.

On suppose dans ce cas que l'espace virtuel est assez grand pour contenir tous les objets en mémoire d'exécution, pour toutes les applications.

III.3.4.3 Couplage statique dans plusieurs espaces virtuels

Les objets sont couplés statiquement (lors de leur création) dans plusieurs espaces virtuels. En général, la liaison statique implique qu'un objet ne sera couplé que dans un seul espace virtuel, ce qui est indépendant du fait que plusieurs espaces virtuels soient gérés. On peut alors distinguer deux cas :

- Un seul objet par espace virtuel

  L'adresse de couplage d'un objet peut être la même quels que soient les objets. Cela signifie alors qu'un espace virtuel différent est géré pour chaque objet. Le contexte de résolution de noms n'a pas besoin d'être géré, puisque tout objet est accédé à la même adresse.

  Cette technique est utilisée dans le système Argus (section III.1.1) dans lequel chaque objet en mémoire d'exécution est couplé dans le domaine de protection et l'espace virtuel privés d'un processus Unix, chaque appel à un objet nécessitant une migration de l'activité vers le domaine et l'espace virtuel contenant l'objet appelé.

  Elle est également utilisée dans le projet Clouds (section III.1.3). Une activité s'exécute dans un processus qui adresse l'objet courant à une adresse fixe à laquelle l'objet est couplé. A chaque appel à un nouvel objet, l'objet courant est découplé et l'objet appelé est couplé à cette même adresse.

  Dans les deux cas, les objets sont de grosse taille ; l'appel de méthode sur un objet est une opération lourde et coûteuse, puisqu'elle nécessite un changement de domaine et d'espace virtuel. C'est pourquoi ces deux systèmes gèrent des objets de plus petite taille, ce qui nous amène à l'autre solution.

- Plusieurs objets par espace virtuel

  Dans ce cas, les objets sont regroupés dans des ensembles logiques. Un espace virtuel est associé à chacun de ces ensembles et à chaque objet de l'ensemble est attribuée statiquement une adresse virtuelle. Un domaine utilisé pour adresser un tel regroupement d'objets est généralement appelé un *serveur d'objets*. Un contexte est associé à ce regroupement et donne pour chaque objet l'adresse qui lui est associée de façon statique.

III.3.4.4 Couplage statique dans un espace virtuel unique

On gère alors un espace virtuel unique dans le système tout entier. L'adresse associée à l'objet lors de sa création est généralement utilisée comme nom de l'objet. Aucun contexte de résolution de nom n'est à gérer, le nom d'un objet donnant directement l'adresse de couplage de l'objet dans cet espace virtuel unique. On suppose alors que tous les objets du système peuvent être contenus dans l'espace virtuel.

Cette technique est utilisée dans les projets Amber (section III.1.6) et Opal (section III.1.7). Dans le système Opal, un objet est accédé dans un domaine en utilisant l'adresse virtuelle que constitue son nom. Un objet peut être couplé dans un domaine de protection sur demande explicite, ou implicitement par le système à la suite d'un adressage invalide.

L'intérêt d'une telle technique est d'éviter l'opération coûteuse de résolution de nom à chaque utilisation d'un nom d'objet, qui aurait lieu si des espaces virtuels différents étaient gérés.

Cette solution est d'autant plus crédible que des processeurs dont les espaces virtuels sont adressés par des adresses de 64 bits arrivent sur le marché.

III.3.5 Comparaison

Nous reprenons ici certains systèmes qui sont révélateurs des compromis qu'il faut réaliser sur les aspects qui ont été cités auparavant, à savoir l'organisation de la mémoire d'exécution pour la protection (confinement), le partage des objets et l'adressage des objets. Ces systèmes sont présentés dans le tableau ci−dessous.

	Taille des objets	**Isolation**	**Partage**	**Adressage**	
				Défaut d'Objet	**Adressage**
Argus	Gros	Par objet	Migration	A chaque appel	Adresse constante
Clouds	Gros	Par objet	Couplage (va−et−vient) et Migration	A chaque appel	Adresse constante
Emerald	Petits	Non	Migration	Au premier appel et au changement de site	Adresse statiquement associée à l'objet
Opal	Petits	Par objet	Couplage (va−et−vient) et Migration	Au premier appel et au changement de domaine	Adresse statiquement associée à l'objet

	Taille des objets	Isolation	Partage	Adressage	
				Défaut d'Objet	Adressage
Gothic	Gros	Non	Couplage (va−et−vient) et Migration	Au premier appel et au changement de site	Adresse statiquement associée à l'objet
Guide−1	Petits	Non	Couplage sur le même site, Migration entre sites différents	Au premier appel et au changement de site	Adresse allouée dynamiquement, Interprété

La première remarque concerne la taille moyenne des objets gérés par ces systèmes. Dans les systèmes Argus et Clouds, les objets sont de grosse taille et l'appel de méthode est une opération coûteuse. C'est pourquoi des objets de plus petite taille sont gérés par les langages de programmation supportés. Dans le système Guide−1, des mesures ont montré que la taille moyenne des objets est de l'ordre de 300 octets. Il apparaît donc nécessaire de fournir un support adapté à la gestion d'objets de petite taille.

Dans un système à objets, il est également nécessaire d'assurer l'isolation des objets entre eux. Dans les projets Guide−1 et Emerald, tous les objets chargés en mémoire d'exécution sur un site sont adressables par tous les processus s'exécutant sur ce site. Si on ne peut pas assurer que tous les langages supportés garantissent l'isolation des objets, cette isolation doit être assurée par le système. Les systèmes Argus et Clouds assurent une isolation par objet, mais les objets sont de grosse taille.

Différentes techniques peuvent être utilisées pour partager des objets entre les processus utilisateurs. Il s'agit essentiellement du partage par migration (rendez−vous) sur la même machine et du partage par couplage avec maintien de la cohérence. Les systèmes les plus récents proposent ces deux stratégies. Il faut utiliser la technique la plus rentable en fonction des besoins des applications.

Enfin, l'appel de méthode doit également être le plus efficace possible. Il peut être pénalisé pour deux raisons : par des changements de domaine de protection fréquents (cela dépend du degré d'isolation fourni) ou par le temps de traduction entre le nom de l'objet et son adresse en mémoire d'exécution. Cette traduction peut être accélérée en associant statiquement une adresse à chaque objet. Cette solution est notamment explorée dans le projet Opal, mais de nombreux problèmes restent à résoudre, notamment la protection (l'isolation et le contrôle d'accès) dans ce type de système.

III.4 Conclusion

Nous avons présenté dans cet état de l'art les différentes approches pour la gestion d'objets partagés persistants dans les systèmes répartis. Cette présentation s'est décomposée en deux parties :

- la gestion de la mémoire de stockage qui sert de support aux objets persistants. Le problème à résoudre à ce niveau est la localisation des objets à partir de leur nom dans cette mémoire constituée des disques des stations de travail. Cette localisation doit prendre en compte le fait que les objets peuvent être déplacés (changer de site de résidence) dans cette mémoire.
- La gestion de la mémoire d'exécution qui contient les objets en cours d'exécution, ainsi que les activités réalisant des appels de méthode sur les objets. L'organisation de la mémoire d'exécution dépend principalement : du degré de protection offert pour l'isolation des objets et des utilisateurs et des techniques utilisées pour réaliser le partage d'objets et l'adressage de ces objets.

C'est dans ce contexte que les chapitres suivants vont décrire le travail réalisé pour la gestion des objets partagés persistants dans le système réparti Guide.

Chapitre IV

Expérience du projet Guide : de Guide−1 à Guide−2

La conception de la version actuelle de Guide, désignée sous le nom de Guide−2, a été précédée, donc influencée, par une première version appelée Guide−1. Il convient donc de présenter les motivations qui ont menées à l'élaboration de cette première version, ainsi que les choix de conception de celle−ci, afin de mieux présenter et argumenter ceux relatifs à la seconde version.

La conception de la seconde version a également été influencée par des évolutions significatives de la technologie et des marchés de l'informatique. On peut citer parmi ces évolutions l'arrivée des micro−noyaux (tels que Mach ou Chorus) pour les évolutions techniques, ou le fait que le langage orienté−objet C++ se soit imposé comme norme (de par l'importance de ses usagers), pour les évolutions du marché.

Ce chapitre présente donc les éléments qui ont influencé la conception de Guide−2. La section IV.1 présente les choix de conception du système Guide−1. Une évaluation critique de cette première version, prenant en compte les évolutions de la technologie et du marché, est proposée en section IV.2. Enfin, les orientations générales de conception de Guide−2 sont présentées et justifiées en section IV.3.

IV.1 Choix de conception de Guide−1

L'objectif du projet Guide [Balter91] est de concevoir et de réaliser un système pour le développement d'applications réparties sur un réseau local de stations de travail et de serveurs. La première phase du projet a été menée de 1987 à 1990.

Le domaine couvert par le système Guide est celui des applications coopératives, caractérisées par une interaction forte entre un ensemble d'utilisateurs, par l'intégration d'applications multiples et par le partage d'informations complexes et à longue durée de vie. Des exemples sont notamment la gestion de documents, le développement de logiciels, ou les outils de communication.

Guide−1 offre principalement :

- Un noyau de système offrant les services essentiels pour le support d'un unique langage (le langage Guide [Krakowiak90]).

  Le modèle à objets est mis en œuvre par un langage permettant la définition de types et de classes avec relation de sous−typage et d'héritage simple. Le noyau fourni dans Guide−1 est spécifiquement conçue pour le support de ce langage.

- Des structures d'exécution appelées *Domaines*(1) contenant des flots d'exécution appelés *Activités.*

  Un Domaine est un espace d'adressage contenant des objets accessibles par des Activités. Des Activités sont des flots d'exécution séquentiels concurrents à l'intérieur d'un même Domaine. Un Domaine est potentiellement réparti et peut être représenté sur plusieurs sites. Par conséquent, les Activités d'un Domaine peuvent s'exécuter en parallèle sur des machines différentes.

  Dans ce modèle, le partage d'objet (associé à des contraintes de synchronisation [Decouchant91]) est l'unique moyen de communication entre des Activités du même Domaine ou de Domaines différents.

- Des objets passifs et persistants liés dynamiquement dans les Domaines.

  Le modèle proposé par Guide est un modèle à objets passifs. Ces objets sont persistants et ils survivent à l'arrêt du système. Ils sont rendus accessibles aux Domaines par une opération appelée *liaison*. Pour qu'une Activité puisse appeler une méthode sur un objet, cet objet doit être lié dans son Domaine.

Pour la réalisation de Guide−1, le système Unix a été choisi comme plate−forme de développement, afin de limiter la phase de développement et d'accroître la portabilité du système.

Nous présentons maintenant les choix de conception de Guide−1 relatifs à la gestion des objets dans le système. Cette présentation est faite en deux parties : la réalisation de la mémoire de stockage et la réalisation de la mémoire d'exécution.

IV.1.1 Mémoire de stockage

La mémoire de stockage, appelée aussi Mémoire Permanente d'Objets (MPO), est le support de conservation persistante des objets et est constituée de l'ensemble des disques du réseau de stations de travail. Comme dans le modèle proposé au chapitre II, la mémoire de stockage est composée d'un ensemble de partitions appelées *systèmes d'objets*. Dans Guide−1, un système d'objets est situé sur un seul site et un objet n'appartient qu'à un seul système d'objets. Un identificateur unique est associé à chaque système d'objets.

La technique utilisée pour le nommage des objets de Guide−1 est d'associer à chaque objet un identificateur logique appelé *Object IDentifier* (*oid*), unique dans le système tout entier. Dans la terminologie du chapitre précédent, Guide−1 utilise donc un nommage absolu des objets. L'*oid* d'un objet est alloué à la création de l'objet et est composé de l'identificateur du système d'objets dans lequel l'objet est créé, ainsi que d'une estampille locale à ce système d'objets. Un *oid* est dépendant de la localisation de l'objet désigné lors de sa création.

Les objets sont désignés par des références systèmes (*SysRef*). Afin de permettre la migration d'objets, une référence à un objet est composée de deux parties (Fig. 4.1). La première partie contient l'*oid* de l'objet référencé et sert à l'identifier de façon unique. La

(1) Afin d'éviter toute confusion avec les termes définis au chapitre III, les abstractions fournies par le système Guide seront écrites avec une première lettre majuscule.

deuxième partie contient une information indiquant la dernière résidence connue de l'objet et est utilisée pour sa localisation en mémoire de stockage. Cette deuxième partie peut être erronée suite à une migration d'objet, mais elle est remise à jour à chaque utilisation. La localisation réelle de l'objet est toujours maintenue à jour dans le système d'objets dans lequel l'objet a été créé. Ainsi, si la deuxième partie d'une référence est périmée, le système d'objets de création est interrogé est donne la localisation réelle, ce qui évite la gestion de liens de poursuite.

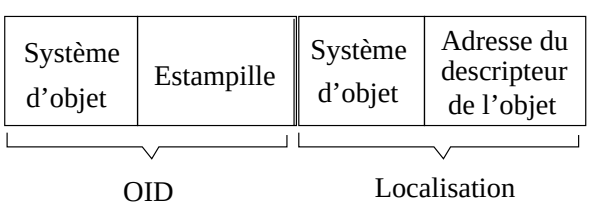

Fig. 4.1 : Format d'une référence système dans Guide−1

La localisation d'un objet en MPO se déroule donc comme suit. On recherche tout d'abord l'objet à l'aide de la partie *Localisation* de la référence. Si on échoue, la partie *OID* est utilisée pour récupérer dans le système d'objets de création la localisation réelle, puis l'objet est localisé. La partie *Localisation* de la référence est dans ce cas mise à jour avec la localisation réelle.

L'espace de stockage est partitionné en *serveurs de MPO*. Un serveur de MPO est responsable de la gestion d'une partie des systèmes d'objets du système. Des tables d'administration associent à chaque système d'objets son serveur de MPO.

IV.1.2 Mémoire d'exécution

Dans Guide−1, les structures d'exécution fournies par le système sont les Domaines et les Activités. Les Activités d'un Domaine partagent le même espace d'adressage, cet espace d'adressage étant composé des objets liés dans ce Domaine. De plus, les espaces d'adressage de Domaines différents doivent pouvoir partager des objets par ce mécanisme de liaison d'objet.

Nous allons présenter respectivement l'organisation de la mémoire d'exécution, la réalisation du partage d'objet et les mécanismes de résolution de nom mis en œuvre lors des appels de méthode.

IV.1.2.1 Organisation de la mémoire d'exécution

Rappelons qu'un Domaine peut être représenté sur plusieurs sites ; nous appelons *Domaine local* la représentation d'un Domaine sur un site.

Pour réaliser le partage d'objets entre des Activités sur le même site, deux solutions étaient envisageables sur Unix :

- Utiliser la mémoire partagée d'Unix.

  Chaque Activité est alors représentée par un processus Unix. Le partage d'objets entre Activités est réalisé en utilisant les mécanismes de mémoire partagée. L'espace d'adressage d'un Domaine local, partagé par les Activités qui s'exécutent à l'intérieur, n'a pas d'existence réelle, puisque la technique de partage entre Activités est la même quels que soient les Domaines d'appartenance de ces Activités.

- Réaliser un noyau de multiplexage de processus légers dans un processus Unix.

  Avec un noyau de multiplexage, des processus légers partagent l'espace virtuel du processus Unix. Deux solutions sont alors envisageables.

  On peut choisir de réaliser un Domaine local par un processus Unix. L'espace d'adressage de ce Domaine local est représenté par l'espace virtuel de ce processus Unix et il est partagé par les Activités de ce Domaine réalisées par des processus légers. Le partage d'objets entre des Domaines locaux différents sur ce site doit être réalisé en utilisant la mémoire partagée d'Unix.

  On peut également choisir avec un tel noyau de multiplexage de réaliser toutes les structures d'exécution d'un site avec un seul processus Unix. Chaque Activité sur ce site est représentée par un processus léger dans ce processus Unix et le partage d'objets entre ces Activités est réalisé de façon implicite par partage de l'espace virtuel du processus entre les processus légers. L'utilisation de mémoire partagée n'est plus nécessaire. Dans cette solution, la notion d'espace d'adressage de Domaine partagé par ses Activités n'a pas d'existence physique.

Lors de la conception de Guide−1, les systèmes Unix disponibles (SunOS, Ultrix, BOS, SCO) n'offraient pas de processus légers. De plus, avec un noyau de multiplexage réalisé dans un processus Unix, une opération d'entrée−sortie aurait bloqué tous les processus légers s'exécutant dans le processus Unix courant. En conséquence, pour la conception de Guide−1, c'est la première solution qui a été choisie. Elle a également l'avantage de minimiser les efforts de développement.

IV.1.2.2 Partage

Chaque Activité est donc réalisée par un processus Unix et l'espace des objets présents en mémoire d'exécution, appelé aussi Mémoire Virtuelle d'Objets (MVO), est constitué de segment(s) de mémoire partagée par les processus Unix. La première Activité désirant utiliser un objet provoque le chargement de cet objet en MVO dans une zone de mémoire partagée.

La gestion du partage des objets avec de la mémoire partagée d'Unix a posé un problème : Unix ne permet pas la gestion d'un grand nombre de segments de mémoire partagée et les performances des opérations de couplage et découplage de segments partagés dans un espace virtuel de processus sont plus que moyennes. Nous avons donc réalisé deux versions successives, l'une avec un segment par objet, l'autre avec un segment unique. Nous présentons ces deux versions :

- Un segment de mémoire partagée par objet.

  Dans la première version, chaque objet amené en MVO sur un site se voit allouer un segment de mémoire partagée sur ce site et y est chargé. Chaque segment de mémoire partagée est désigné par une clé qui sert à le désigner de façon externe pour demander son couplage.

  Un contexte de résolution de noms est donc géré sur ce site et associe à chaque nom d'objet présent en MVO la clé désignant le segment de mémoire partagée contenant l'objet et permettant son couplage par une Activité. Ce contexte est réalisé par une table appelée Table de la Mémoire Virtuelle d'Objets (TMVO) qui est partagée par tous les processus Unix sur ce site.

  Un autre contexte doit également être géré dans chaque processus pour indiquer pour un nom d'objet d'une part s'il a été couplé dans l'espace virtuel de ce processus et d'autre part pour donner l'adresse de couplage de l'objet, un objet pouvant être couplé à des adresses différentes dans des processus différents. Ce contexte est réalisé par une table locale au processus.

  Cette solution, qui est conceptuellement satisfaisante (car l'unité de couplage est l'objet), a dû être abandonnée pour les raisons de performance énoncées précédemment, mais également parce que la taille minimale d'un segment de mémoire partagée est une page, ce qui est bien supérieur à la taille moyenne de nos objets.

- Un segment de mémoire partagée par site.

  Dans la version actuelle de Guide−1, un segment de mémoire virtuelle est alloué sur chaque site à l'initialisation du système pour contenir tous les objets chargés en MVO sur ce site. Ce segment est couplé à la même adresse par chaque processus Unix sur ce site qui partage donc la MVO entière du site.

  Un seul contexte de résolution de noms est nécessaire dans ce cas pour gérer le chargement et l'adressage des objets. Ce contexte peut être partagé par tous les processus du site, étant donné que le chargement d'un objet entraîne son partage à la même adresse pour tous les processus. Ce contexte de résolution est la même TMVO que dans le cas précédent, mis à part l'identification externe du segment de mémoire partagée contenant l'objet qui est remplacée par l'adresse de l'objet, valable pour tous les processus du site.

Dans les versions des systèmes Unix utilisées au début du développement de Guide−1, le partage de mémoire virtuelle n'était possible que sur une même machine. En conséquence, on n'autorise le partage d'objet en mémoire d'exécution que sur un seul site à un instant donné. Si une Activité désire partager un objet déjà chargé dans la MVO d'un autre site, ce partage est réalisé par changement de site d'exécution de cette Activité. Ce mécanisme est appelé *extension* de l'Activité vers un site distant et prend la forme d'un appel de procédure à distance (RPC).

Le mécanisme d'extension est réalisé en créant des représentants d'une même Activité sur plusieurs sites (un par site). Une Activité qui doit s'étendre vers un site contacte un représentant sur ce site ou en crée un s'il n'y en a pas. Le schéma d'appel à distance est

synchrone. Un seul représentant s'exécute à un instant donné et les autres attendent le prochain appel à distance.

IV.1.2.3 Résolution de noms

On a vu en section IV.1.2.2 qu'un contexte de résolution de noms d'objets appelé TMVO était géré sur chaque site et partagé par toutes les Activités du site. Pour réaliser l'adressage des objets, c'est à dire obtenir l'adresse de couplage d'un objet à partir de son nom, la solution par défaut consiste à consulter la TMVO à chaque utilisation d'objet.

A cette solution s'ajoutent des raccourcis d'adressage permettant d'améliorer l'efficacité de l'appel de méthode. Comme dans la plupart des systèmes, le principal raccourci d'adressage de Guide−1 consiste à modifier à la première utilisation d'une référence à un objet, le mode d'utilisation de celle−ci afin que les prochaines utilisations soient plus efficaces.

Dans le langage Guide, une référence langage vers un objet n'a pas le même format qu'une *SysRef*. Une référence langage (*LgeRef*) est composée d'une *SysRef* augmentée d'une information sur la localisation de l'objet en mémoire d'exécution (Fig. 4.2).

Fig. 4.2 : Format d'une référence langage dans Guide−1

Cette information contient initialement une valeur indiquant que la référence n'a pas encore été utilisée. A la première utilisation, il y a résolution du nom donné par la partie *SysRef*, dans le contexte de nommage (TMVO) du site. Si l'objet n'est pas présent dans la MVO de ce site et s'il n'est dans aucune MVO, alors il est chargé dans la MVO locale et la TMVO est mise à jour. S'il est dans une MVO distante, il y a extension de l'Activité vers le site dont la MVO contient l'objet.

Si l'objet désigné par cette référence langage est ou peut être chargé localement, alors l'information sur la localisation de l'objet en MVO dans la référence langage est mise à jour. Cette information aurait pu être l'adresse de l'objet, mais c'est en fait l'index de l'objet dans la TMVO. Le fait que ce soit l'index permet tout d'abord le vidage d'objet de la MVO, mais aussi de stocker d'autres informations de localisation dans l'entrée associée à l'objet dans la TMVO (l'index en TMVO de la classe de l'objet notamment[(2)]).

Cette gestion de l'adressage des objets est illustrée sur la figure Fig. 4.3. Deux Activités sur le même site partagent tous les objets chargés dans la MVO du site. L'objet de *SysRef SR_1* contient une référence (*LgeRef*) vers l'objet de *SysRef SR_2*. Cette référence langage contient l'index i (dans la TMVO) de l'objet référencé, permettant de mettre en œuvre un adressage plus rapide.

Nous décrivons maintenant la réalisation d'un appel de méthode.

---

(2) Dans Guide−1, une classe est gérée comme un objet.

Dans Guide−1, tout appel de méthode en mémoire d'exécution est compilé en un appel à une primitive du système : *guideCall (...)*. Un bloc d'appel de méthode contenant la description de l'appel est passé en paramètre. Ce bloc contient notamment :

- L'adresse de la *LgeRef* de l'objet appelé. La partie *SysRef* peut être mise à jour si l'objet a été déplacé. La partie contenant l'index dans la TMVO peut être mise à jour si elle ne l'était pas.
- Le nom de la méthode appelée.
- L'adresse de la zone de paramètres.
- Le nombre de paramètres.

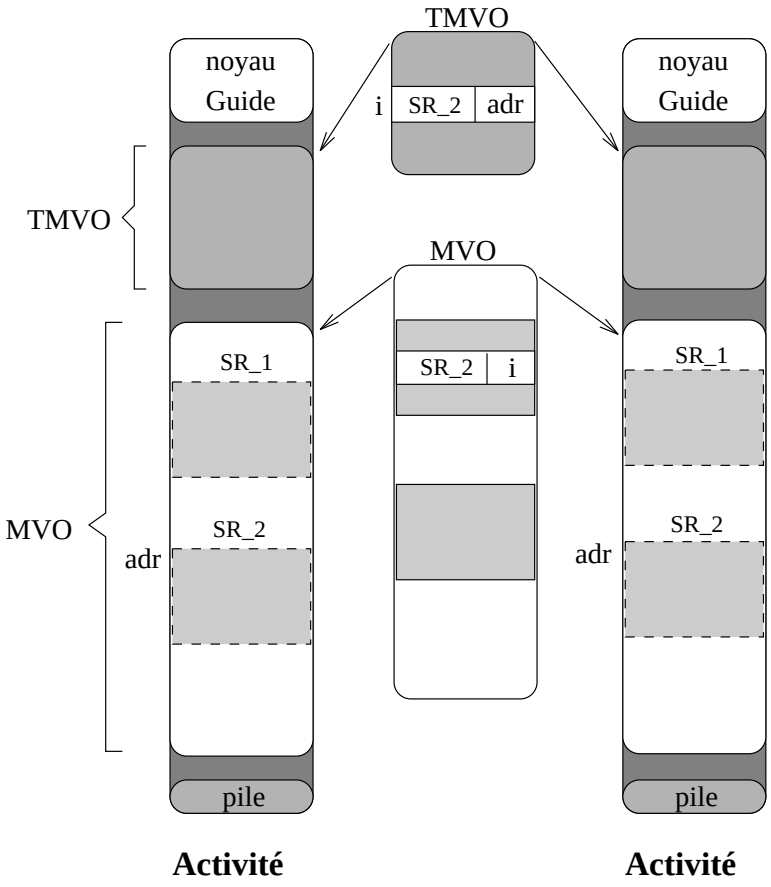

Fig. 4.3 : Adressage des objets dans Guide−1

Dans le langage Guide, la méthode appelée est toujours recherchée dans la classe de l'objet appelé. Un objet contient une référence vers sa classe. Une classe contient le code des méthodes qu'elle définit et une table de méthodes ou Table des Définitions Externes (TDE), qui indique pour chaque méthode son nom, le déplacement du code de la méthode dans la classe, ou la référence de la classe contenant le code de la méthode si la méthode est héritée.

La primitive *guideCall* est alors composée des étapes suivantes :

- Localisation de l'objet appelé et de sa classe en MVO.

  Rappelons que si l'index de l'objet en TMVO est à jour dans la *LgeRef*, la localisation de l'objet et de sa classe en MVO est immédiate. Dans le cas contraire, une recherche dans la TMVO est exécutée. Cette opération peut provoquer le chargement de l'objet dans la MVO locale, ou l'extension de l'Activité vers un autre site.
- Recherche de la méthode appelée.

  La méthode est recherchée dans la TDE de la classe. Si la méthode est héritée d'une superclasse, cette classe doit être localisée en MVO et la méthode recherchée dans sa TDE.
- Appel effectif de méthode.

Un autre mécanisme d'accélération de l'appel de méthode a été mis en place afin d'améliorer les appels successifs de la même méthode. Un cache de méthode est géré dans le système : un appel de méthode est identifié par la référence de la classe et le nom de la méthode. Si l'appel courant est trouvé dans ce cache, on évite la phase de recherche de la méthode dans la classe et dans une superclasse.

IV.1.2.4 Conclusion

En résumé, les caractéristiques de la mémoire d'exécution du système Guide−1 sont les suivantes :

- Les Activités sont réalisées par des processus Unix.
- Le partage sur un même site est réalisé par chargement des objets dans un grand segment de mémoire partagée. Aucun type d'isolation n'est assuré, que ce soit entre les Activités ou entre les objets.
- Le partage entre sites différents est réalisé par un mécanisme d'extension des Activités (RPC). Deux Activités ne peuvent partager un objet que sur le même site.
- La résolution des noms d'objet est optimisée en ajoutant une information sur la localisation de l'objet en mémoire d'exécution (l'index de l'objet en TMVO) dans chaque référence à un objet.
- Tout appel de méthode se traduit par un appel à une primitive système réalisant cet appel. On dit que les appels de méthode sont interprétés.

IV.2 Evaluation critique de Guide−1

Une évaluation complète de Guide−1 peut être trouvé dans [Freyssinet91a]. Nous présentons seulement les éléments servant à expliquer les choix de conception de Guide−2.

IV.2.1 Langages

Le choix de base dans Guide−1 a été celui d'une intégration forte entre langage de programmation et système d'exploitation. Le système sert de support d'exécution pour des applications écrites dans un langage unique (Guide). Dans le domaine des langages de programmation orientés−objets, le langage C++ s'est imposé comme une norme. Bien que le langage Guide soit un réel succès, il s'est avéré primordial d'être capable de fournir une extension persistante et distribuée du langage C++ sur notre système. De plus, fournir un noyau de système permettant le support de plusieurs langages est un défi scientifique intéressant.

IV.2.2 Mémoire d'exécution

Les critiques que l'on peut formuler sur la gestion de la mémoire d'exécution de Guide−1 concernent quatre aspects : la gestion des structures d'exécution, le partage des objets, l'appel de méthode et la protection.

Structures d'exécution

Les Activités sont réalisées par des processus Unix. Leur gestion (création, destruction, extension) en est donc alourdie. De plus, ces Activités ne partagent pas de façon implicite l'espace d'adressage de leur Domaine, comme cela aurait été le cas si elles avaient été réalisées par des processus légers partageant le même espace virtuel d'un processus. Les Domaines n'ont dans Guide−1 qu'une existence conceptuelle.

L'émergence des micro−noyaux permet une réalisation plus adaptée du partage dans notre modèle. Sur Mach [Accetta86], les processus (*tâches*) sont des espaces d'adressage dans lesquels plusieurs flots d'exécution (*threads*) peuvent s'exécuter en concurrence. Un représentant local de Domaine peut être réalisé par une *tâche* dont les *threads* constituent les extensions d'une ou plusieurs Activités qui partagent implicitement les objets présents dans l'espace d'adressage de la tâche.

Partage

Tout représentant d'Activité sur un site partage tout objet présent dans la MVO sur ce site. Cette réalisation de la MVO n'est pas satisfaisante, car elle a pour conséquence une absence totale de protection entre les objets et entre les Activités, tout objet de la MVO étant potentiellement adressable par toute Activité sur ce site.

La création d'un segment de mémoire partagée par objet n'était pas une solution possible, étant donné la taille moyenne de nos objets et le coût de l'opération de couplage. Il est toutefois possible de gérer des regroupements d'objets que nous appelons *grappes* et d'associer un segment de mémoire partagée à chaque grappe lors de son couplage sur un site. Ainsi, chaque Activité n'a accès qu'aux grappes qu'elle a couplées et le coût d'un couplage est amorti dès que plusieurs objets sont utilisés dans la grappe. Un tel regroupement d'objets peut également servir à améliorer les performances des opérations d'entrée−sortie en permettant le chargement de plusieurs objets en même temps.

Les micro−noyaux tels que Mach ou Chorus [Rozier88] apportent une technologie qui facilite la gestion de grappes persistantes. Ils offrent la possibilité de définir des serveurs qui répondent aux fautes de pages sur les segments de mémoire partagée. Un tel outil peut permettre le chargement des données d'une grappe à la demande (page par page). Ces serveurs de pagination permettent également le partage de segments entre des machines différentes, ce qui permet la mise en œuvre de nouvelles techniques de partage.

Appel de méthode

Chaque appel de méthode est réalisé par appel à la primitive *guideCall*. Cette primitive teste la liaison de l'objet et de sa classe et recherche la méthode à exécuter dans la hiérarchie des classes à partir de la classe de l'objet. Cette technique est onéreuse et il doit être possible de mettre en place un mécanisme au premier appel qui évite d'entrer dans le système Guide pour les appels suivants. L'appel de méthode coûte dans Guide−1 environ 100 fois plus que l'appel de procédure. Il paraît possible de réduire ce fossé.

Protection

Comme nous l'avons décrit dans le chapitre II, le terme de protection prend un double sens : il signifie non seulement la nécessité d'isoler les Activités et les objets contre les erreurs des programmes ou contre des malveillances des programmeurs, mais également celle de permettre à tout utilisateur de protéger les applications créées en spécifiant les droits d'accès des autres utilisateurs à ses propres objets, en termes de méthodes appelables sur ces objets.

- Dans Guide−1, aucune isolation n'est assurée, toute erreur (ou malveillance) dans une méthode pouvant écraser tous les objets présents en MVO, dans le grand segment de mémoire partagée. L'isolation des Activités et des objets repose donc sur la sûreté(3) du langage Guide.

  La supposition de sûreté des langages supportés ne peut plus être faite dans le cas du langage C++, car ce langage permet d'adresser directement par des pointeurs l'espace virtuel courant. Il faut définir et réaliser des mécanismes limitant la portée d'adressage d'une Activité exécutant un appel de méthode sur un objet.
- Les aspects relatifs au contrôle d'accès dans Guide−1 n'ont pas été traités. Ils doivent être pris en compte dans la conception du système Guide−2.

IV.2.3 Mémoire de stockage

Le système doit être capable de désigner un grand nombre d'objets dans l'espace et dans le temps. Dans Guide−1, une référence système est composée de 64 bits :

- un identificateur unique sur 32 bits,
- une partie localisation courante en MPO sur 32 bits.

---

(3) La confiance dans le code généré par le compilateur Guide.

Dans une nouvelle version, on veut être capable de désigner des objets à l'aide d'identificateurs de 64 bits. L'extension des références à 128 bits pour permettre la migration d'objets n'est pas acceptable, car augmentant fortement la taille moyenne des objets sur disque. Il faut donc prévoir un schéma de désignation n'augmentant pas la taille des identificateurs d'objets et permettant la migration des objets.

IV.3 Orientations générales de Guide−2

Cette section résume les orientations générales de conception du système Guide−2, tenant compte de l'évaluation critique précédente de Guide−1.

Machine virtuelle multi−langage

Le système Guide−2 est destiné à fournir un noyau que nous appelons **Eliott** pour le support de différents langages comme les langages Guide et C++. Il faut donc définir une interface générique, permettant l'élaboration des modèles complexes des langages orientés−objets devant être supportés.

Modèle d'exécution

Il reste le même que dans Guide−1. Il est fondé sur les notions de Domaines et d'Activités. La nouvelle réalisation prévoit une réelle existence de la notion d'espace d'adressage de Domaine, donc une isolation physique de ces espaces d'adressage.

Gestion des objets

La gestion de la mémoire en deux niveaux MPO et MVO reste un choix de base. Les objets sont regroupés en *Grappes* d'objets et la grappe est l'unité de couplage en MVO. Le transfert des données entre la MPO et la MVO est réalisé à la demande lors des fautes de pages.

Désignation des objets

Les objets sont désignés de façon uniforme en MPO et en MVO par des références systèmes (*SysRef*) de 64 bits. Un mécanisme doit être prévu pour permettre la migration.

Liaison des objets

Les appels de méthode ne sont plus interprétés systématiquement. Le mécanisme de liaison au premier appel évite les interprétations lors des appels suivants.

Protection

Deux types de problèmes doivent être résolus :

- L'isolation
    - Des Domaines

      Une erreur dans l'exécution d'un programme ne doit pas perturber les autres exécutions en cours (du même usager ou d'autres usagers).

- Des objets

  Nous ne faisons pas la supposition que tous les langages supportés sont sûrs. Il faut donc limiter les facultés d'adressage (les objets accessibles) d'une Activité s'exécutant dans un objet.

- Le contrôle d'accès

  Il doit permettre à un usager de spécifier les droits d'accès accordés aux autres usagers sur les objets dont il est propriétaire. Pour être cohérent avec le modèle d'objet, ces règles de protection doivent être définies en terme de méthodes applicables sur les objets.

Conception sur le micro−noyau Mach

Le choix du micro−noyau Mach 3.0 pour la réalisation du noyau Eliott se justifie par une meilleur adéquation des fonctions fournies par rapport à Unix et plus particulièrement :

- Processus légers

  Plusieurs flots d'exécution peuvent s'exécuter en concurrence dans le même espace d'adressage.

- Gestion mémoire

  Il est possible de gérer la pagination des segments de mémoire partagée.

- Communications

  Les adresses de destination de message cachent les aspects liés à la répartition et les droits d'utilisation de ces adresses de message sont contrôlés par le noyau Mach.

Le chapitre suivant présente les principes de conception de la gestion des objets dans le noyau Eliott.

Chapitre V

Principes de conception du noyau Eliott

Dans le chapitre précédent ont été présentées les motivations pour la réalisation de la seconde version du système Guide. Le but de ce chapitre est de présenter ses principes de conception.

L'objectif de cette version est de fournir un noyau pivot que nous appelons **Eliott** [Rousset93] pour le support de plusieurs langages orientés−objets. Nous définissons tout d'abord dans la section V.1 en quoi consiste ce noyau pivot, puis nous présentons les principes de réalisation du support d'objets partagés persistants dans ce noyau, en distinguant comme dans les chapitres précédents la gestion de la mémoire de stockage (section V.2) et la gestion de la mémoire d'exécution (section V.3). Enfin la section V.4 présente la conception de ce noyau en utilisant ces principes.

V.1 Un noyau pivot

Le but du noyau Eliott est de fournir les mécanismes minimaux permettant l'élaboration de modèles à objets et de langages plus complexes [Chevalier92b] ; il sert donc de pivot pour les environnements d'exécution de ces langages. Les langages visés sont actuellement le langage Guide conçu lors de la première phase de notre projet et une extension du langage C++. L'utilisation de ce noyau comme plate−forme pour le support d'autres langages orientés−objets pourra être envisagée ultérieurement. La figure Fig. 5.1 décrit l'architecture proposée.

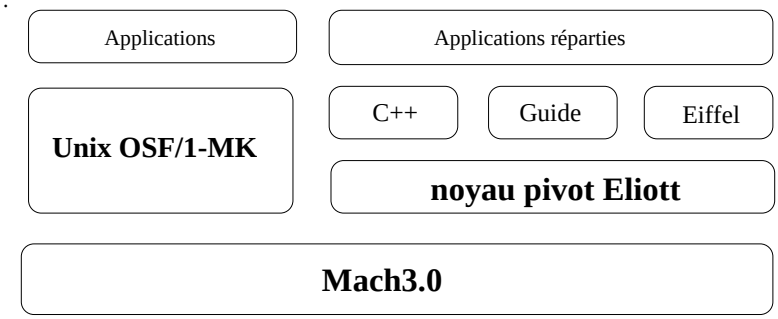

Fig. 5.1 : Architecture de l'environnement Guide

Afin d'être en mesure de réaliser les divers modèles de ces langages, nous avons défini un modèle à objets primitif qui ne retient que les aspects essentiels des modèles fondés sur la notion d'objet. Ce modèle primitif est sans héritage ; les modèles d'héritage doivent être réalisés par les compilateurs des langages.

Dans ce modèle, les principales abstractions définies sont les **instances**, les **classes** et les **librairies** de code. Ces entités sont des objets persistants et désignés par une référence système unique dans le système.

Les instances et les classes sont des entités séparées. Un objet de type instance est un exemplaire d'une seule et unique classe et contient une référence vers cet objet de type classe. La seule façon d'accéder à un objet est d'appeler une méthode sur cet objet. Les méthodes sont définies dans sa classe.

Un objet de type classe est un descriptif des méthodes définies dans la classe. Le code des méthodes est stocké dans des objets de type librairie. Un objet de type classe contient donc un ensemble de références externes vers les objets de type librairie contenant le code des méthodes définies dans la classe.

Ce modèle primitif à objets est représenté sur la figure Fig. 5.2. L'objet instance contient une référence vers sa classe. Cette classe définit deux méthodes *m1* et *m2*. Une référence externe dans l'objet classe est associée à chaque méthode. Cette référence externe désigne le code de la méthode dans l'objet librairie qui la contient.

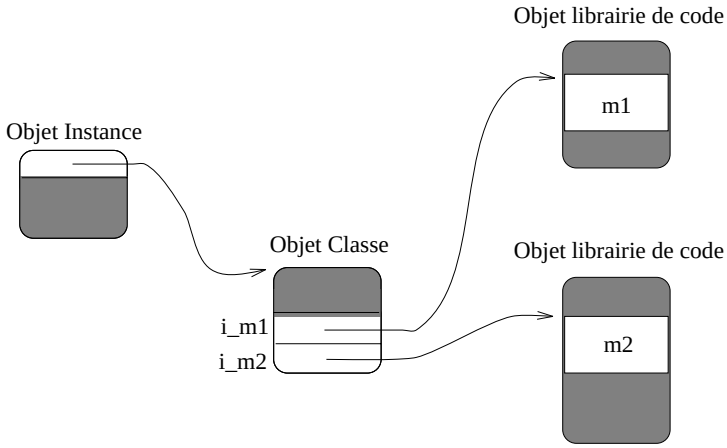

Fig. 5.2 : Le modèle à objets primitif d'Eliott

Dans ce modèle, un objet instance peut contenir une référence vers un autre objet instance.

On peut remarquer qu'une référence d'un objet classe vers un objet librairie n'est pas seulement un nom d'objet, mais plutôt un nom d'objet accompagné d'un déplacement dans la librairie de code, correspondant au point d'entrée de la méthode désignée. Une référence externe dans un segment est donc composée d'un nom d'objet et d'un déplacement dans cet objet.

V.2 Gestion de la mémoire de stockage

V.2.1 Organisation

La mémoire de stockage est composée de **Volumes** et de **Grappes**.

Un volume est une zone de données assimilable à une partition de disque. C'est une entité d'administration. Le fait de rajouter ou supprimer un volume est une opération manuelle réalisée par l'administrateur du système. Un volume est de taille fixe et contient un ensemble de grappes. Le nombre de grappes dans un volume est variable et ces grappes sont allouées dynamiquement à la demande (de la mémoire d'exécution).

Une grappe est une zone de données dans un volume dans laquelle sont gérés des objets. La grappe est l'unité de partage entre les structures d'exécution. C'est donc l'unité de couplage dans les tâches gérées en mémoire d'exécution. La grappe est une notion qui peut être cachée au programmeur, le système gère alors le regroupement des objets dans les grappes. Elle peut également être exploitée par les langages de programmation pour permettre au programmeur de définir sa propre politique de regroupement.

La figure Fig. 5.3 illustre cette architecture de la mémoire de stockage.

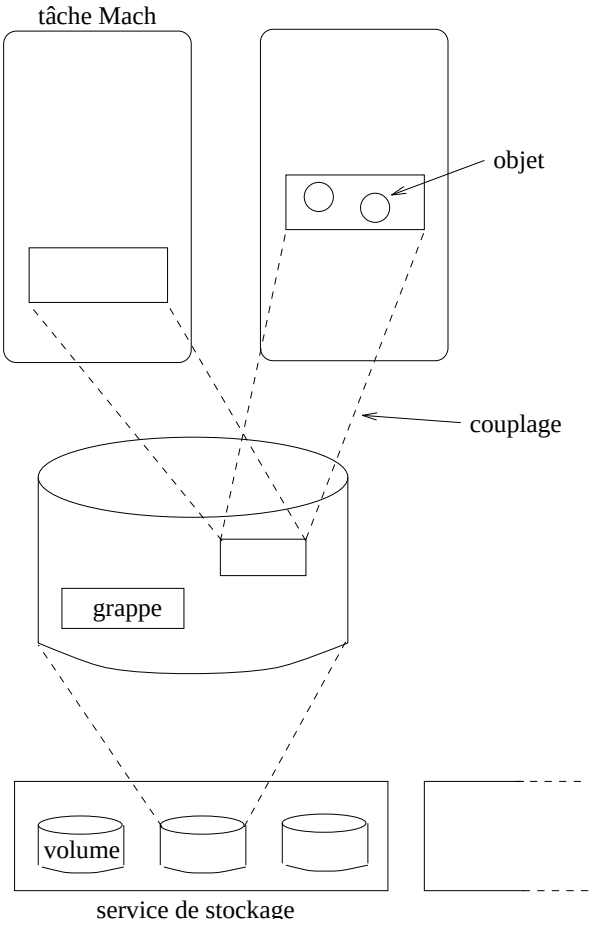

Fig. 5.3 : Organisation de la mémoire de stockage

Chaque volume est géré par un **service de stockage**. Un service de stockage est une entité pouvant être répartie, assurant la persistance des volumes (donc des grappes) et permettant la lecture et l'écriture de leurs données lors des transferts vers la mémoire d'exécution. Des exemples de services de stockage sont le système de fichier d'Unix (avec des montages NFS) ou un ensemble de serveurs dupliqués pour être tolérants aux pannes [Chevalier92a].

Les grappes sont rendues accessibles en mémoire d'exécution par couplage dans des tâches de Mach (gérées en mémoire d'exécution). Le transfert des données de ces grappes entre la mémoire de stockage et la mémoire d'exécution est réalisé à la demande lors des fautes de pages dans les grappes. Lors d'une faute de page dans une grappe couplée, le service de stockage gérant le volume contenant cette grappe est consulté.

Les objets sont gérés dans les grappes. Le couplage d'un objet d'une grappe implique le couplage de tous les objets de cette grappe.

Un des traits important est la recopie des grappes à la fin de l'exécution entre la mémoire d'exécution et la mémoire de stockage. La recopie d'une grappe est réalisée lorsqu'elle n'est plus utilisée en mémoire d'exécution et elle ne concerne que les pages qui ont été modifiées. La recopie d'une grappe est faite de façon atomique.

V.2.2 Désignation

Les objets sont désignés par une référence système (*SysRef*) de 64 bits. Cette *SysRef* est unique dans tout le système (c'est un nom absolu). Elle est allouée à la création de l'objet et elle est dépendante de la localisation de l'objet lors de sa création. La *SysRef* d'un objet est composée de trois champs (Fig. 5.4) :

- Un identificateur de Volume (*vol_id*).
- Un identificateur de Grappe (*clu_id*)[(2)]
- Un identificateur de l'objet dans cette grappe (*obj_id*)

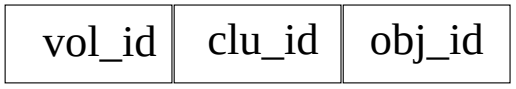

Fig. 5.4 : Référence système d'un objet

Le choix de ce schéma de désignation des objets se justifie par la supposition que les migrations d'objets seront rares. Cette désignation permet alors une localisation très rapide pour la grande partie des objets qui ne sont jamais déplacés.

---

(2) *clu* est l'abréviation pour *cluster*, le terme anglais pour grappe.

V.2.3 Localisation et migration

La technique utilisée par défaut pour localiser un objet consiste à coupler dans la tâche courante sa grappe de création donnée par sa *SysRef* et à rechercher l'objet dans cette grappe. Une grappe doit être couplée pour que l'on puisse tester la présence d'un objet à l'intérieur.

L'unité de migration est l'objet, la migration de grappes entre volumes n'est pas un objectif. Le but de la migration d'objets est de permettre le changement de grappe de résidence de l'objet, soit pour agir sur la gestion de cet objet en mémoire de stockage (certaines grappes peuvent être dupliquées pour assurer une plus grande disponibilité des données), soit pour regrouper des objets dans les grappes pour profiter d'un effet de localité lors du couplage. Dans ce deuxième cas, la technique utilisée pour la migration des objets doit répondre aux objectifs suivants :

- Elle ne doit pas faire payer un surcoût trop élevé pour la localisation de l'objet déplacé, puisque la migration a pour but la diminution du coût de cette localisation par regroupement d'objets dans la même grappe.
- Elle ne doit pas augmenter le coût de la recherche des objets qui ne sont pas déplacés.
- Elle ne doit pas augmenter la taille des références utilisées pour désigner les objets.

Ces objectifs éliminent les solutions énumérées au chapitre III. On élimine notamment la solution qui consiste à gérer un lien de poursuite dans la grappe de création de l'objet, indiquant la grappe de résidence de l'objet, car on ne veut pas payer le coût du couplage de la grappe de création de l'objet pour la localisation d'un objet qui se trouve dans une autre grappe qui est déjà couplée.

Nous avons choisi de réaliser la migration des objets à l'aide de catalogues de migration stockés dans les grappes et de liens de poursuite. Cette solution repose sur les hypothèses et les contraintes suivantes :

- L'opération de couplage est coûteuse. Des mesures sont données au chapitre VIII.
- Le parcours de liens de poursuite ou la consultation de catalogues dans des grappes couplées coûte peu, relativement à l'opération de couplage.
- Etant donné que les appels de méthodes ne sont pas toujours interprétés (cet aspect est détaillé en section V.3.3), il n'est pas possible de connaître l'identité de la grappe courante, c'est à dire la grappe contenant l'objet appelant dans l'appel de méthode qui nécessite la localisation de l'objet appelé. Cette contrainte nous interdit donc de gérer un catalogue dans chaque grappe pour consulter le catalogue de la grappe courante au cours du processus de localisation.

Dans notre solution, un catalogue qui enregistre les objets déplacés (arrivés) dans la grappe est géré dans chaque grappe. Lorsqu'un objet change de grappe de résidence, un lien de poursuite est laissé dans la grappe *origine* et cette migration d'objet est enregistrée dans le catalogue de migration de la grappe *destination.*

Dans chaque tâche, un catalogue des objets déplacés couplés dans la tâche est géré. Ce catalogue est enrichi à chaque couplage de grappe, avec le catalogue des objets déplacés (arrivés) dans la grappe. Ce catalogue est en fait une liste chaînée des catalogues de migration des grappes.

Lorsqu'un objet doit être localisé, on parcourt les liens de poursuite laissés dans les grappes à partir de la grappe de création de l'objet, tant qu'aucun couplage de grappe n'est nécessaire. Si aucun couplage n'est requis, l'objet est localisé.

Si la chaîne des liens de poursuite passe par une grappe non couplée, alors le catalogue des objets déplacés couplés dans la tâche est consulté, ce qui permet de détecter si l'objet est déjà couplé (on évite un couplage inutile). Si l'objet n'est pas trouvé dans ce catalogue, alors la grappe est couplée et on continue avec les liens de poursuite.

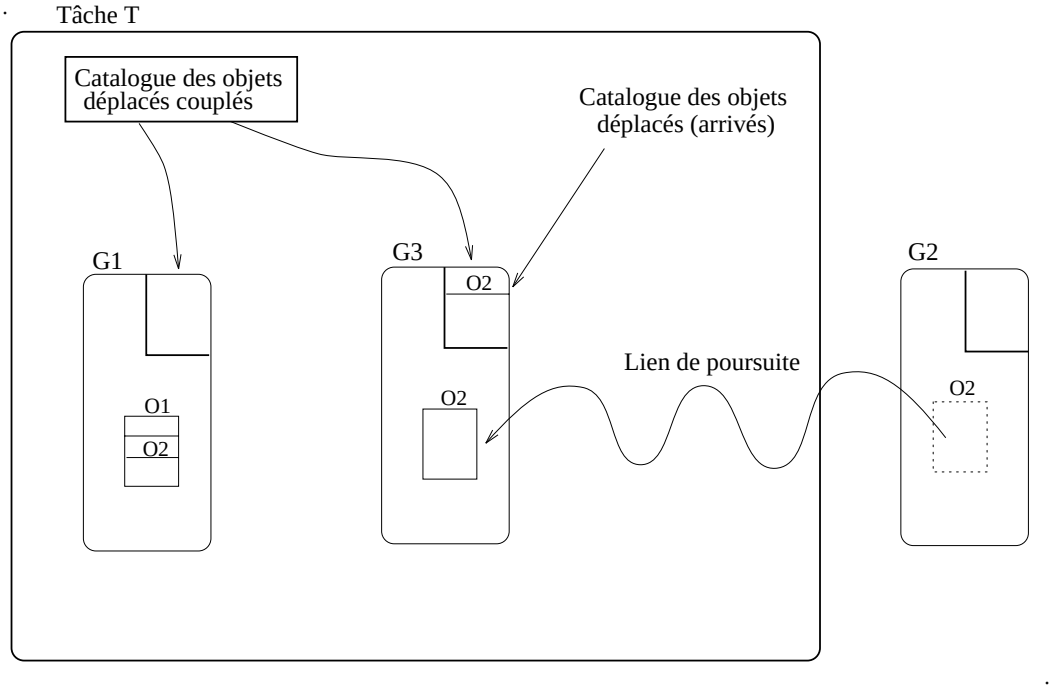

Fig. 5.5 : Mécanisme de migration d'objets

Cette solution est illustrée sur la figure Fig. 5.5. Un objet *O1* dans une grappe *G1* contient une référence à un objet *O2*. L'objet *O2*, initialement dans la grappe *G2*, a été déplacé dans la grappe *G3*. Un lien de poursuite a été laissé dans *G2* et *O2* a été enregistré dans le catalogue des objets arrivés dans *G3*. Les grappes *G1* et *G3* étant couplées, le catalogue des objets déplacés couplés dans la tâche désigne les catalogues des grappes couplées. On peut alors envisager deux cas dans la localisation de *O2* :

- La grappe *G2* est couplée.

  Dans ce cas, la localisation va être effectuée à partir de la grappe de création de *O2* (*G2*), puis le(s) lien(s) de poursuite va être suivi pour arriver à *O2* dans *G3*.

- La grappe *G2* n'est pas couplée.

  Le catalogue des objets déplacés couplés est consulté et on y trouve la localisation de l'objet *O2*.

L'avantage de cette solution est d'éviter le couplage inutile de grappes si l'objet recherché est déjà couplé. Par contre, si les grappes *G2* et *G3* ne sont pas couplées, alors il faudra les coupler toutes les deux pour localiser *O2*.

On peut remarquer que l'impact sur les objets non déplacés est négligeable. Il y aura un surcoût dû à la recherche dans le catalogue pour la première localisation d'objet dans une grappe non couplée, mais toutes les localisations suivantes dans cette grappe ne seront pas pénalisées.

Lorsque les liens de poursuite sont utilisés, il est possible de mettre à jour le lien de poursuite dans la grappe de création avec la localisation réelle de l'objet.

Discussion

Parmi les contraintes qui nous ont amenés à choisir cette solution, il y a le fait que notre mécanisme d'appel de méthode, qui n'interprète pas tous les appels, ne permet pas de connaître lors de la localisation la grappe contenant l'objet appelant. Cette contrainte est très gênante, car elle élimine des solutions qui auraient été très efficaces. Nous en décrivons brièvement deux, qui diffèrent par l'utilisation des catalogues de migration.

La première solution consiste à gérer (comme dans notre solution) dans chaque grappe un catalogue des objets arrivés dans la grappe. Lorsque l'objet *O1* appelle l'objet *O2* et lorsque *O2* doit être localisé, si la consultation du catalogue est nécessaire (grappe non couplée), c'est le catalogue dans la grappe de *O1* qui est consulté, ce qui permet de détecter une migration qui a réuni l'objet appelant et l'objet appelé. L'inconvénient de cette approche est qu'elle ne donne pas la grappe de localisation d'un objet déplacé, si l'objet se trouve dans une grappe autre que la grappe contenant l'objet appelant. Par contre, on limite la taille du catalogue consulté.

La seconde solution consiste à gérer dans chaque grappe un catalogue donnant la localisation des objets référencés depuis des objets dans la grappe, lorsque les objets référencés ont été déplacés. Comme il n'est pas possible de mettre à jour lors d'un déplacement tous les catalogues des grappes contenant des références vers cet objet, un tel catalogue est géré comme un cache et mis à jour de façon paresseuse. Si le catalogue ne contient pas la localisation de l'objet, on accepte le couplage de la grappe de création de l'objet. Si le catalogue contient une localisation, ce n'est pas forcément la bonne, mais le catalogue est alors mis à jour pour les prochaines localisations. Par rapport à la solution précédente, celle−ci permet d'accélérer également la localisation des objets déplacés dans une grappe différente de la grappe courante.

V.2.4 Récapitulatif

La mémoire de stockage est composée de Volumes contenant des Grappes. Le Volume est une partition de disque et la grappe est l'unité de couplage dans une tâche.

Les noms d'objets sont dépendants de la grappe de création de l'objet. La localisation est directe pour un objet non déplacé.

Le mécanisme de localisation d'un objet déplacé utilise :

- Un lien de poursuite dans la grappe de création de l'objet.
- Un catalogue par grappe contenant les références des objets déplacés présents dans cette grappe.
- Un catalogue des objets déplacés dans chaque tâche enrichi à chaque couplage avec le catalogue de la grappe couplée.

V.3 Gestion de la mémoire d'exécution

V.3.1 Organisation de la mémoire d'exécution

Le but de l'organisation des structures d'exécution est d'une part de réaliser les Domaines et les Activités de Guide et d'autre part de garantir l'isolation des objets et des activités décrite au chapitre III. L'isolation des activités signifie dans le cas de Guide l'isolation entre des Activités appartenant à des Domaines différents.

Pour réaliser un représentant local de Domaine (Domaine local), il suffit d'utiliser les tâches de Mach permettant l'exécution de processus légers (*threads*) dans l'espace virtuel associé à cette tâche. Chaque représentant d'Activité dans ce Domaine local est représenté par un processus léger de Mach dans cette tâche. Une telle réalisation assure l'isolation des Activités, puisque les facultés d'adressage d'une Activité sont réduites à l'espace d'adressage de la tâche réalisant le Domaine local.

L'isolation par objet n'est pas possible, étant donné que l'unité de couplage est la grappe. L'isolation aurait pu être réalisée au niveau de la grappe en couplant chaque grappe dans une tâche différente, mais cela aurait nécessité un changement de tâche pour chaque appel entre des objets de grappes différentes.

Notre réalisation de l'isolation entre objets consiste à interdire (à l'intérieur d'un Domaine) le couplage dans la même tâche d'objets appartenant à des propriétaires différents. Ainsi, un Domaine peut être composé sur une même machine de plusieurs tâches associées à des propriétaires d'objets différents. Pour accèder à un objet, il faut le coupler dans une tâche associée au propriétaire de cet objet, en créant une tâche associée à ce propriétaire s'il n'y en a pas. Un appel de méthode entre des objets appartenant à des propriétaires différents est réalisé par un mécanisme d'extension de l'Activité, pour transférer le flot d'exécution d'une tâche à l'autre.

L'organisation des structures d'exécution est décrite sur la figure Fig. 5.6. Le Domaine *D2* est représenté sur les deux sites. Sur le site 2, *D2* est composé de tâches associées aux propriétaires *X*, *Y* et *Z*. Sur le site 1, *D2* n'est composé que d'une tâche associée à *X*. Une grappe est partagée entre les deux Domaines *D1* et *D2*. L'Activité *A2*, après des appels à des objets de *Z* et de *Y* nécessitant un changement de tâche, a réalisé un appel à distance sur un objet de *X*.

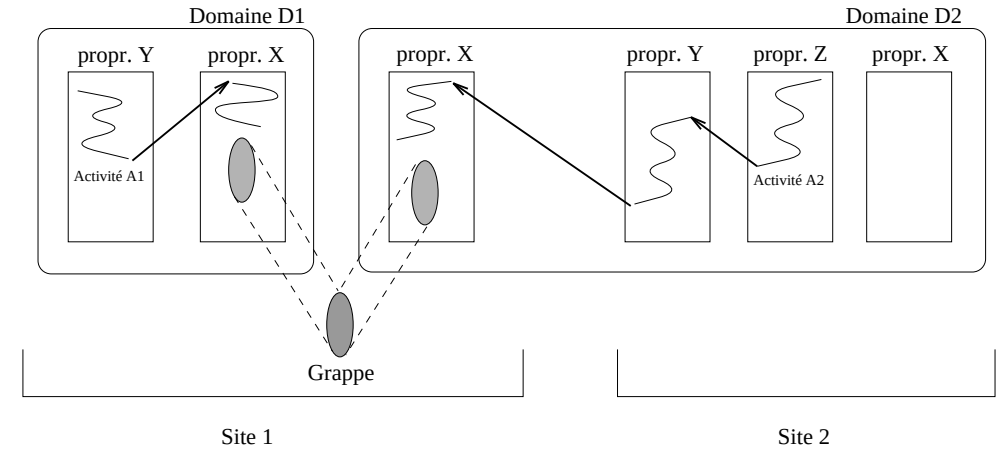

Fig. 5.6 : Organisation de la mémoire d'exécution

Ce choix a deux implications :

- Tous les objets d'une même grappe doivent appartenir au même propriétaire, puisque la grappe est l'unité de couplage.
- Etant donné qu'empiriquement, un objet nouvellement créé l'a toujours été dans le domaine de protection courant, cela signifie qu'un objet créé appartient au propriétaire associé à la tâche courante, donc au même propriétaire que le propriétaire de l'objet demandant la création.

On peut opposer à cette solution celle qui consiste à faire appartenir l'objet créé à l'usager pour le compte duquel le Domaine s'exécute. Notre solution nous paraît bien plus cohérente. En effet, si on considère l'exemple d'un texte composé de chapitres et de sections (tous ces objets appartenant à l'usager *X*), l'opération appelée par l'usager *Y* qui ajoute une section dans un chapitre va s'exécuter dans une tâche associée à *X* et va logiquement créer une section appartenant à *X* et non à *Y*. Cette solution est également utilisée dans les projets Birlix [Kowalski90] et Melampus [Richardson92].

Cette organisation de la mémoire d'exécution assure donc l'isolation :

- Des Activités.

  Puisque les Domaines ne partagent pas de tâches, une Activité d'un Domaine ne peut adresser que l'ensemble des grappes couplées dans les tâches composant son Domaine.
- Des objets au niveau des propriétaires d'objets.

  L'exécution d'une méthode sur un objet ne peut adresser (par une erreur ou une malveillance) que des objets appartenant au même propriétaire.

V.3.2 Partage

Le partage est réalisé par couplage dans les tâches. La possibilité de définir des serveurs de pagination sur le micro−noyau Mach permet le partage de grappes entre machines différentes.

Pour réaliser le partage de grappes entre des Activités sur des machines différentes, il existe donc deux techniques qui sont le couplage (réparti) et le mécanisme d'extension (changement de site d'exécution par RPC).

On utilise le couplage réparti pour les grappes en lecture seule ou rarement modifiées, afin de bénéficier de la possibilité de lecture en parallèle sur plusieurs machines.

L'utilisation de l'extension permet de se ramener à un partage local à un site pour les grappes souvent modifiées, et d'éviter le coût du maintien de la cohérence qui peut provoquer un fort trafic de pages sur le réseau.

Le choix de la technique utilisée pour le partage d'une grappe dépend actuellement d'un attribut associé statiquement à la grappe lors de sa création. Dans une prochaine étape, il pourra être envisagé de faire ce choix dynamiquement en fonction du type des accès à la grappe partagée. On retrouve ces deux mécanismes de mise en œuvre du partage dans le projet Clouds [Dasgupta90]. Le choix du mécanisme utilisé est réalisé dynamiquement.

V.3.3 Résolution de nom

Pour la résolution de nom dans le noyau Eliott, deux solutions étaient envisageables :

- Associer statiquement (à la création) une adresse différente à chaque grappe.

  Cette approche est notamment explorée dans les projets Opal [Chase92b] et Lucas [Inohara93]. Une grappe est alors toujours couplée à la même adresse dans toutes les tâches la partageant. Il est alors nécessaire de disposer de grands espaces virtuels. L'arrivée des espaces virtuels à 64 bits rend plausible une telle solution. Cette solution a été écartée, puisque nous ne disposons pas encore de ce type de machines. Elle est actuellement étudiée par d'autres thésards dans notre équipe.
- Associer dynamiquement une adresse à une grappe lors de son couplage dans un espace virtuel.

  C'est la solution que nous avons choisie.

On peut alors choisir pour une grappe de la coupler à la même adresse dans toutes les tâches la partageant. Cette solution a l'avantage de permettre l'utilisation de la technique de mutation de pointeur décrite au chapitre III. Il faut alors synchroniser l'allocation des adresses virtuelles de couplage entre toutes les tâches, ce qui peut être très coûteux.

Nous avons donc choisi de permettre le couplage d'une grappe à des adresses virtuelles différentes dans différents espaces virtuels de tâches. Le même objet peut donc être couplé à des adresses différentes. Par abus de langage, on parlera dans le suite de couplage d'objet, même s'il s'agit en fait du couplage de la grappe contenant l'objet.

Un contexte de résolution de nom est donc géré dans chaque tâche et associe à chaque objet couplé son adresse de couplage. Pour accélérer cette résolution de nom, une approche segmentée à la Multics [Daley68] a été adoptée.

Chaque objet est réalisé par un segment. Un segment est une zone de données contiguës pouvant contenir (comme un objet) un nom de segment et nous nous intéressons donc à la résolution de ce nom à l'exécution, sachant que ces segments peuvent être couplés à des adresses différentes.

Dans chaque tâche dans laquelle un segment est couplé, une zone de mémoire privée à cette tâche est associée au segment et contient toutes les données dépendant de l'espace virtuel de cette tâche, concernant ce segment. Les données dépendant de cet espace virtuel sont généralement des adresses virtuelles. Cette zone est appelée *segment de liaison* du segment.

Un segment de liaison est géré comme une table dont une entrée est allouée pour chaque nom de segment contenu dans le segment. Cette entrée donne notamment l'adresse de couplage dans l'espace virtuel de la tâche du segment référencé, s'il y est couplé.

L'identification d'un segment dans l'espace virtuel d'une tâche est alors constituée de deux adresses virtuelles : l'adresse de couplage du segment et l'adresse de son segment de liaison permettant l'adressage accéléré des segments référencés depuis ce segment. Cette double adresse est appelée *vecteur d'accès* du segment dans l'espace virtuel courant. Dans la suite de cette description, nous adoptons la notation suivante : le premier champ d'un vecteur d'accès s'appelle *s* (pour *state*) et le deuxième champ s'appelle *ls* (pour *linkage section*).

A chaque référence externe dans un segment est associé un indice désignant une entrée dans le segment de liaison. Cette entrée contient l'identification du segment référencé dans l'espace virtuel courant, donc elle contient un vecteur d'accès.

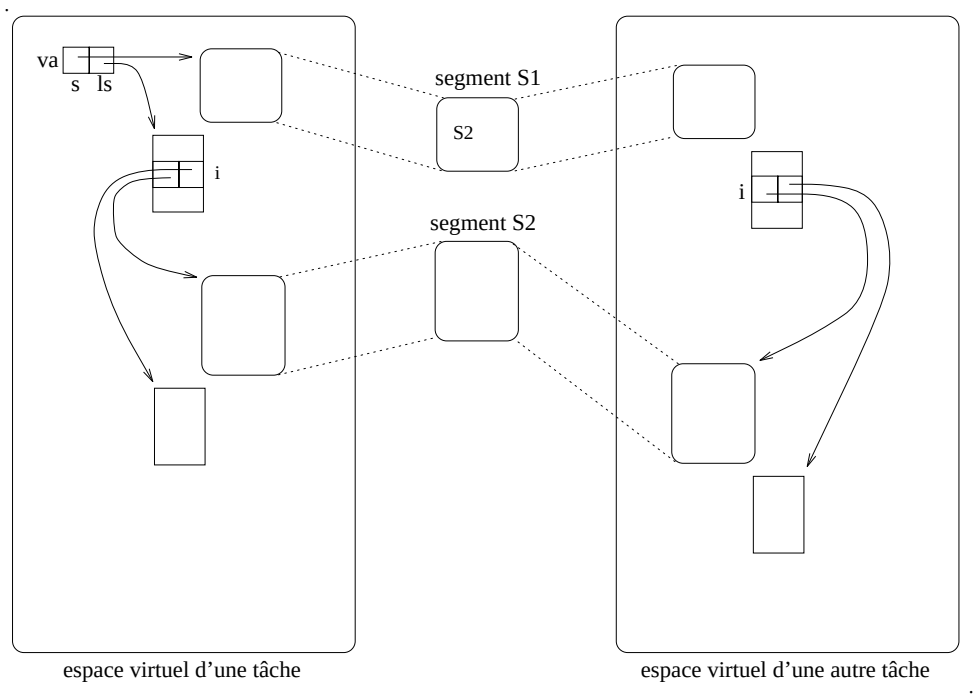

Fig. 5.7 : Une approche segmentée

Ce schéma est illustré sur la figure Fig. 5.7. Si un vecteur d'accès **va** identifie un segment (*S1*) et si ce segment contient une référence externe d'indice **i**, alors l'identification du segment référencé dans l'espace virtuel courant est **va.ls[i]** et l'adresse du segment référencé est **va.ls[i].s**.

Le segment de liaison est alloué à la première utilisation du segment dans l'espace virtuel. Il est initialisé avec des valeurs nulles. On appelle *liaison d'une référence* l'opération qui consiste à mettre à jour le vecteur d'accès associé à une référence externe dans un segment. Chaque utilisation d'un vecteur d'accès vérifie donc si les valeurs de ce vecteur sont nulles et demande la liaison de la référence associée si c'est le cas. La liaison d'une référence consulte le contexte de résolution de nom de l'espace virtuel, couple le segment si nécessaire et met à jour le vecteur d'accès.

Pour les segments dont les données peuvent être modifiées en cours d'exécution (les instances), l'affectation d'une référence à un segment nécessite en principe l'invalidation de l'entrée associée à cette référence dans tous les segments de liaison existants. Cette solution a été remplacée par la suivante : le test de liaison d'une référence vérifie également si le vecteur d'accès dans le segment de liaison désigne le segment désigné par la référence utilisée. Cela nécessite que le segment désigné ait en en−tête son identification. Ce test n'est nécessaire que pour les segments modifiables.

V.4 Conception du noyau pivot

Nous présentons maintenant la conception du noyau pivot Eliott qui utilise les principes de segmentation décrits auparavant [Freyssinet91b].

Dans le modèle à objets primitif fourni, chaque entité (instances, classes et librairies) est réalisée par un segment. C'est pourquoi on utilisera également les termes segment d'instance, segment de classe et segment de code.

Une instance est donc un segment contenant une référence vers son segment de classe. Un segment d'instance peut contenir d'autres références externes à des segments d'instance. Un indice désignant une entrée dans le segment de liaison de ce segment est alors statiquement (lors de la définition de la classe de l'objet) associé à chaque référence vers un autre segment. L'adressage d'un objet désigné par une référence externe utilise le vecteur d'accès identifié par cet indice. La première référence externe d'un segment d'instance est la référence à sa classe ; l'indice 0 lui est attribué.

Une classe est un segment contenant une référence externe par méthode définie dans la classe. Un indice désignant une entrée dans le segment de liaison du segment de classe est statiquement (à la compilation de la classe) associé à chaque méthode. L'adressage d'une méthode utilise le vecteur d'accès désigné par cet indice.

Enfin, une librairie de code est un segment contenant le code des méthodes et des procédures. Un segment de code peut contenir une référence externe vers un autre segment de code, correspondant à un appel de procédure. Un vecteur d'accès est alors également associé à cette référence externe.

La figure Fig. 5.8 illustre cette réalisation.

Le segment d'instance *O1* contient une référence vers le segment de classe *C1* dont le code est stocké dans le segment *Lib1*. Dans *O1,* une référence externe *x* est affectée à la *SysRef* de l'objet *O2.* A cette variable *x* est associé le vecteur d'accès d'indice *i_x* dans le

segment de liaison de *O1*. Lorsque cette référence externe est liée, le champ *s* désigne l'état de *O2* et le champ *ls* le segment de liaison de *O2*.

L'objet *O2* contient une référence vers sa classe, dont le vecteur d'accès est à l'indice 0 dans son segment de liaison.

La classe *C2* définit 3 méthodes *m1*, *m2* et *m3*. Le segment de la classe *C2* contient une référence externe par méthode, vers le segment de code *Lib2* contenant le code des méthodes. Un vecteur d'accès dans le segment de liaison du segment de classe est associé à chaque méthode, respectivement aux indices *i_m1*, *i_m2* et *i_m3*. Chacune de ces références externes vers le code des méthodes donne le déplacement (*d1*, *d2* et *d3*) de la méthode dans le segment de code.

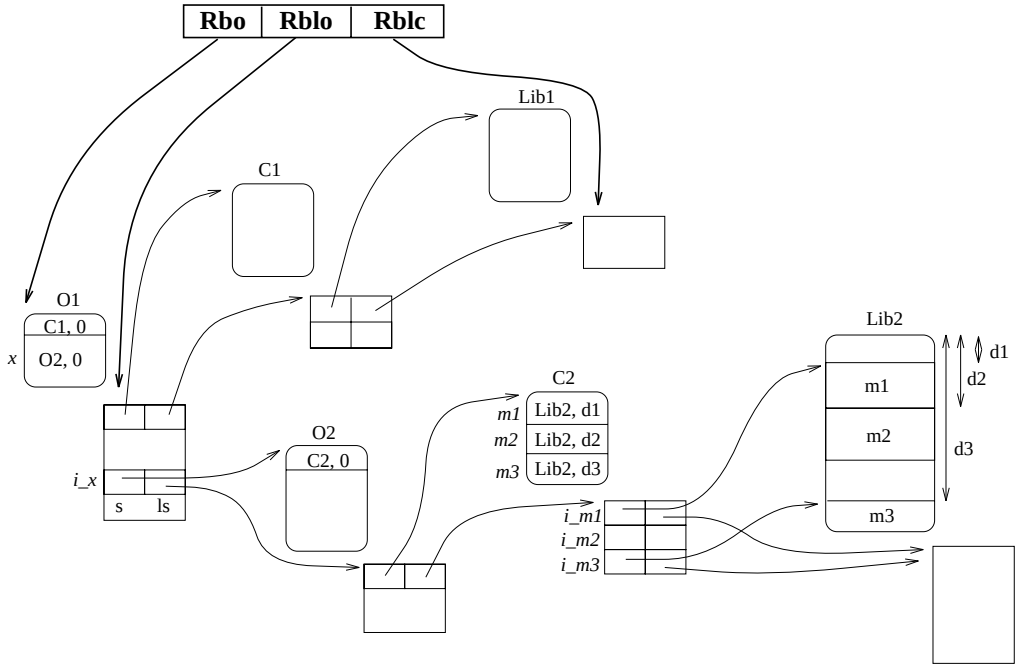

Fig. 5.8 : Réalisation du modèle à objets primitif

Pour réaliser les appels de méthodes, nous gérons trois registres de type pointeur :

- **Rbo** (Registre de Base de l'Objet)

  Ce registre désigne l'état de l'objet courant. Il correspond au premier champ du vecteur d'accès de l'objet courant. Il permet d'accéder aux variables d'état de l'objet courant.

- **Rblo** (Registre de Base de Liaison de l'Objet)

  Ce registre désigne le segment de liaison de l'objet courant. Il correspond au second champ du vecteur d'accès de l'objet courant. Il permet l'utilisation des références externes de l'objet courant.

- **Rblc** (Registre de Base de Liaison du Code)

  Ce registre désigne le segment de liaison du segment de code contenant le code en cours d'exécution.

Ces registres sont commutés à chaque appel de méthode sur un objet.

En théorie, pour adresser différentes séquences de code dans le segment de code courant, il est nécessaire de gérer un registre Rbc (Registre de Base du Code), toute adresse dans le segment de code courant étant un déplacement par rapport à ce registre. En fait, nous faisons la supposition que le code exécuté est indépendant de sa position dans un espace virtuel. Cela signifie que tout adressage interne à ce segment de code est généré relativement au compteur ordinal.

Dans cet exemple, si une méthode est en cours d'exécution sur l'objet *O1* et si le code de cette méthode exécute l'appel de méthode *x.m3*, alors le code généré pour obtenir l'adresse de la méthode appelée est le suivant : Rblo[i_x].ls[0].ls[i_m3].s

Il reste alors à vérifier les liaisons des segments intervenant dans le processus d'appel de méthode avant l'appel de méthode effectif. Pour que l'appel de méthode soit réalisable, il faut que les trois liaisons suivantes soit résolues :

- La liaison de la référence de l'objet appelant vers l'objet appelé.
- La liaison de l'objet appelé vers sa classe.
- La liaison de la classe de l'objet appelé vers le segment de code contenant la méthode appelée.

Lorsque l'environnement d'exécution détecte qu'une liaison doit être réalisée, il doit transmettre au système les paramètres de cette liaison, à savoir :

- La référence à lier, composée d'un identificateur de segment et d'un déplacement dans ce segment. Elle est stockée dans l'état du segment qui contient la référence. Ce segment peut être une instance, une classe ou une librairie de code.
- L'adresse du vecteur d'accès à mettre à jour. Ce vecteur d'accès se trouve dans le segment de liaison du segment contenant la référence et il est désigné par un indice calculé à la compilation du code utilisant les segments.

La réalisation du noyau est détaillée dans le chapitre suivant.

V.5 Conclusion

Ce chapitre a présenté les principes de conception du noyau Eliott.

Pour la gestion de la mémoire de stockage, les principales caractéristiques sont :

- L'organisation en volumes et en grappes, le volume étant une unité d'administration et la grappe l'unité de couplage entre la mémoire de stockage et la mémoire d'exécution.
- La désignation des objets, dépendant du volume et de la grappe de création de l'objet, fondée sur un identifiant unique de 64 bits.
- Un mécanisme de migration fondé sur des catalogues de migration gérés au niveau des grappes et consultés lorsqu'un couplage est nécessaire.

La gestion de la mémoire d'exécution se caractérise par :

- Des Domaines composés de tâches potentiellement réparties. L'isolation des Domaines est assurée en interdisant le partage de tâches entre Domaines. L'isolation des objets est assurée par propriétaire d'objets, en interdisant le couplage d'objets de propriétaires différents dans la même tâche.
- Le partage réalisé par couplage de grappes dans des tâches de Domaines différents. Le mécanisme d'extension permet de réunir des Activités sur un même site pour localiser le partage.
- Une résolution de nom fondée sur une simulation de machine segmentée à la Multics, permettant le partage de grappes entre tâches à des adresses différentes.

Ce noyau pivot fournit le support de plusieurs langages orientés−objets. Il offre les notions nécessaires à l'élaboration de modèles plus complexes : classes, instanciation, partage de code entre classes, appel de méthode.

Des mécanismes permettant le contrôle d'accès ont été intégrés à ce noyau. Ils sont présentés dans le chapitre suivant.

Chapitre VI

Contrôle d'accès dans le noyau Eliott

Le but de ce chapitre est de présenter et justifier les principes de réalisation du contrôle d'accès dans le noyau Eliott. L'objectif du contrôle d'accès est de permettre au programmeur de spécifier les droits qu'il accorde aux utilisateurs sur les objets manipulés par l'application développée.

Nous ne prenons pas en compte ici les problèmes relatifs à la sécurité. Nous supposons que les machines connectées au réseau utilisent toutes le noyau Eliott et que les entités contre lesquelles on veut se protéger sont des applications de ce même noyau.

Après une description des objectifs que doivent atteindre les mécanismes permettant de mettre en œuvre le contrôle d'accès aux objets (section VI.1), nous présentons les différentes approches traditionnellement utilisées dans les systèmes d'exploitation (section VI.2) et enfin notre proposition pour le noyau Eliott (section VI.3).

VI.1 Objectif du contrôle d'accès

L'objectif du contrôle d'accès est de fournir aux applications un moyen de contrôler les droits qu'elles accordent sur les objets qu'elles gèrent. La protection des objets prend toute son importance dans un contexte où les objets sont partagés. Les objets peuvent non seulement être partagés entre les différents utilisateurs d'une application, mais également dans le cadre de différentes applications à des fins de coopération.

Les objectifs généraux qui doivent être pris en compte sont les suivants :

- Un contrôle de l'accès **au niveau de l'objet**.

  Tout objet étant potentiellement partageable, il doit être possible de définir des règles de protection différentes pour chaque objet du système.

- Un contrôle **en fonction de la méthode** appelée sur l'objet.

  Pour obtenir une meilleure intégration avec le modèle à objet, la protection ne doit pas être définie en terme d'opérations de lecture−écriture sur des données, mais en terme de méthodes appelables sur des objets.

- Une protection variable **en fonction du processus** réalisant l'appel.

  La protection doit pouvoir varier en fonction de l'identité du processus appelant la méthode, en général en fonction de l'utilisateur pour le compte duquel le processus s'exécute.

- Une protection variable **en fonction du contexte** (la situation) de ce processus appelant.

  Etant donné qu'un utilisateur peut appeler une même méthode dans le cadre d'applications différentes, il doit être possible d'autoriser ou interdire cet appel de méthode en fonction du contexte de l'appel. Cela implique qu'il doit être possible de gérer différents domaines de protection et de faire varier le pouvoir d'un processus en fonction du domaine dans lequel il s'exécute.

  Ce contrôle a généralement pour but de permettre l'écriture de sous−systèmes protégés. Si on prend l'exemple d'un sous−système dont le but est de fournir un service en interdisant aux clients du service d'appeler directement les objets internes, il faut imposer le passage par des points d'entrée (appelés *guichets*) et n'autoriser les appels aux objets internes que depuis des objets *guichets*.

Il faut donc être en mesure de contrôler les méthodes autorisées sur les objets et de le faire en fonction d'attributs associés au processus (en général l'utilisateur) et au contexte de l'appel (le domaine de protection d'où provient l'appel).

De plus, la réalisation de ces mécanismes de protection doit prendre en compte certaines contraintes de sureté. Les mécanismes de protection fournis doivent en particulier résoudre de façon satisfaisante les problèmes de **méfiance mutuelle** fréquents dans les systèmes d'exploitation. Nous illustrons ce problème de méfiance mutuelle de la façon suivante. Supposons qu'un utilisateur U1 possède un objet O1 et que cet objet soit partagé avec un utilisateur U2, alors il faut assurer les exigences suivantes :

- U2 ne peut appeler qu'une méthode autorisée sur O1.
- Les objets de U2 doivent être protégés du code de la méthode appelée sur O1.

Après avoir passé en revue les exigences concernant le contrôle de l'accès aux objets, nous allons étudier les techniques utilisées dans des systèmes caractéristiques.

Dans un système, les règles de protection des objets peuvent être représentées par une matrice [Lampson71] indicée respectivement par les usagers et par les objets du système (Fig. 6.1). La case *M [User2 , Obj3]* de la matrice donne alors les droits accordés à l'utilisateur *User2* sur l'objet *Obj3*.

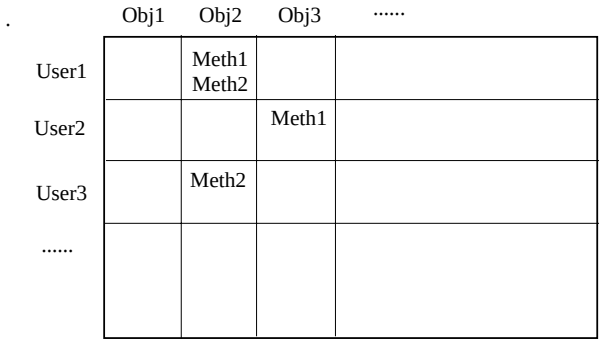

Fig. 6.1 : Matrice de droits

Une approche classique consiste alors à associer à chaque objet une **liste d'accès** qui contient pour chaque usager du système ses droits d'accès sur cet objet. Cette solution revient à regrouper les informations de protection de la matrice par colonnes. Une autre approche classique consiste à associer à chaque usager une liste appelée **liste de capacité**, contenant les droits d'accès sur les objets du système accordés à cet utilisateur. Cette solution revient à regrouper les informations de la matrice par ligne.

Nous présentons maintenant ces approches.

VI.2 Différentes approches

VI.2.1 Protection par listes d'accès

VI.2.1.1 Principes

Une liste d'accès est associée à chaque objet et définit pour chaque usager ses droits en terme d'opérations sur cet objet. Le système a alors la charge de vérifier que seuls des appels autorisés par les listes d'accès sont exécutés.

Ainsi, les trois premiers objectifs sont atteints :

- Il y a contrôle d'accès au niveau de l'objet, puisqu'une liste d'accès est associée à chaque objet.
- Il y a contrôle d'accès en fonction de la méthode et du processus (l'usager qui lui est associé), puisqu'ils servent à définir la liste d'accès d'un objet.

Un mécanisme est généralement ajouté pour permettre un contrôle de l'accès en fonction du contexte du processus appelant. Nous en décrivons ici différents à travers des exemples de systèmes dont la protection est fondée sur des listes d'accès.

VI.2.1.2 Protection dans le système Multics

Multics [Organick72] est un système développé sur une machine spécifique, dont le contrôle d'accès est fondé sur les notions de liste d'accès et d'anneau.

L'entité de base de Multics est le segment. Il y a des segments de données et des segments procédures. Une liste d'accès est associée à chaque segment et donne pour chaque usager du système ses droits sur le segment (lecture, écriture, exécution).

Le mécanisme permettant un contrôle en fonction du contexte du processus appelant est fondé sur la notion d'anneau de protection. Les anneaux de Multics sont concentriques et au nombre de 8. Un processus s'exécute à un instant donné dans un anneau et à chaque segment est associé une suite d'anneaux consécutifs (une *parenthèse*) dans lesquels le segment peut être utilisé. Il y a une parenthèse pour chaque opération (lecture, écriture et exécution) et les parenthèses de lecture et écriture commencent à l'anneau 0 (ce qui implique qu'un processus s'exécutant dans l'anneau 0 peut lire et écrire dans tous les segments). Un processus ne peut utiliser un segment que s'il s'exécute dans un anneau inclus dans la parenthèse de ce segment.

Pour changer d'anneau de protection, un processus doit appeler une opération privilégiée (*CALL*) d'appel de procédure sur un segment procédure se trouvant dans l'anneau cible. A chaque segment procédure est associée une liste de points d'entrée appelés *guichets* et seuls ces points d'entrée sont appelables par l'instruction *CALL*. Il est possible d'associer à ce segment procédure une parenthèse d'appel (différente de la parenthèse d'utilisation) définissant les anneaux depuis lesquels ce segment peut être appelé par CALL. Si un processus s'exécutant dans l'anneau *i* appelle un *guichet* d'un segment procédure de parenthèse *j* avec *i>j* (Fig. 6.2), alors il faut que *i* soit inclus dans la parenthèse d'appel de ce segment. Le processus exécute alors le code de la procédure appelé dans un anneau de la parenthèse *j* (celui de plus faible droit).

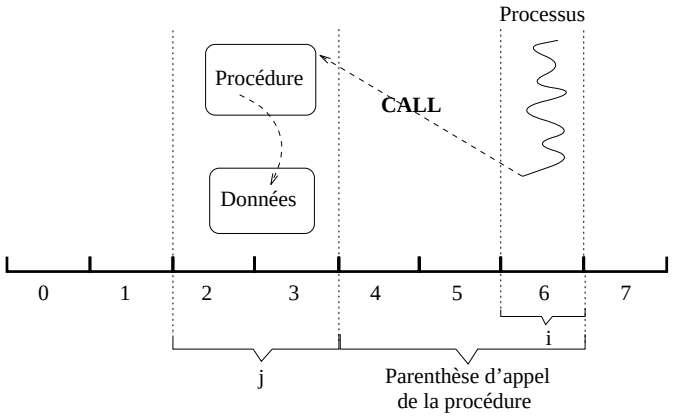

Fig. 6.2 : Les anneaux de Multics

Dans l'exemple du client−service, il suffit d'associer le même anneau d'utilisation à tous les objets réalisant ce service et de faire s'exécuter les processus clients dans un anneau de plus faible droit. On spécifie les points d'entrée des segments procédures et que ces segments peuvent être appelés depuis les anneaux supérieurs. Les segments procédures internes à l'application ont une parenthèse d'appel vide. Les segments de données du service ne peuvent être manipulés que dans l'anneau associé au service.

Multics permet donc avec ses anneaux de faire varier le pouvoir d'un processus en fonction de son contexte. Un processus qui change d'anneau change de domaine de protection et voit ainsi ses droits redéfinis. Cette solution n'est toutefois pas satisfaisante. En effet, si ce mécanisme permet de construire un sous−système protégé contre ses clients parce qu'il s'exécute dans un anneau de plus fort privilège, il ne permet pas de construire des sous−systèmes coopérants mutuellement méfiants. Le sous−système qui sera dans un anneau supérieur ne sera pas protégé contre les actions de l'autre.

VI.2.1.3 Autres solutions de contrôle de l'appelant

D'autre méthodes existent pour contrôler l'accès aux objets en fonction du contexte du processus appelant. Ces méthodes consistent souvent à spécifier dans les listes d'accès une règle de protection dépendant d'un attribut de l'objet appelant comme son propriétaire, sa

classe ou son nom. C'est le cas notamment dans le système Birlix [Kowalski90] dans lequel une entrée d'une liste d'accès peut contenir un identificateur de classe et une méthode, ce qui signifie que l'appel de cette méthode est autorisée pour tous les objets de cette classe.

Ainsi, sur l'exemple du client−service, on positionnera la protection des objets internes de telle sorte qu'ils ne puissent être appelés que depuis un objet de classe *guichet*.

VI.2.1.4 Conclusion

Les listes d'accès sont utilisées dans de nombreux systèmes d'exploitation. Elles facilitent la gestion des informations de protection, car elles sont centralisées au niveau de chaque objet et elles facilitent également la révocation de droit.

Le problème est généralement de fournir une technique permettant un contrôle d'accès en fonction du contexte d'exécution du processus.

VI.2.2 Protection par capacités

VI.2.2.1 Principes

Conceptuellement, une capacité [Levy84] est un ticket permettant à son possesseur d'utiliser un objet avec des droits déterminés. Une capacité contient le nom de l'objet sur lequel des droits sont accordés, ainsi que la définition de ces droits en termes d'opérations sur l'objet. Une opération peut être ici une méthode de l'objet, ou plus simplement une opération de lecture−écriture. Une capacité est protégée par le système dans le sens où il n'est pas possible de la forger ou de la modifier. Les seules opérations permises sur les capacités sont l'exécution d'une opération sur l'objet désigné si elle est autorisée, la diminution de droits et le passage en paramètre d'une opération.

En général, la notion de domaine est introduite pour traduire la variation du pouvoir d'un processus sur les objets du système. Un domaine de protection est un ensemble de droits sur des objets ; c'est donc ici une liste de capacités. Un processus s'exécutant dans ce domaine n'a accès qu'aux objets désignés par une capacité de ce domaine. Pour que l'enceinte de protection définie par un domaine soit réelle, il faut contrôler la façon d'entrer dans un domaine, en associant à un domaine des points d'entrée. Un point d'entrée sur un domaine est une capacité de ce domaine pouvant être utilisée depuis l'extérieur. Lorsqu'une capacité sur un point d'entrée est utilisée par un processus pour appeler une méthode, le processus change de domaine de protection pour exécuter cet appel.

De cette façon, les objectifs généraux cités auparavant sont atteints :

- Un contrôle d'accès **au niveau de l'objet** est réalisé, puisqu'on peut contrôler quelles sont les capacités exportées.
- Un contrôle d'accès **en fonction de la méthode** est réalisé, car une capacité contient une description des méthodes autorisées.
- Le contrôle de l'accès **en fonction du processus** est réalisé, puisqu'on contrôle à qui les capacités sont tranmises.

- Pour permettre de contrôler l'accès aux objets **en fonction du contexte** du processus appelant, il suffit de contrôler la diffusion des capacités vers les différents domaines de protection.

  Ainsi, dans l'exemple du client−service précédemment énoncé, ce service peut être réalisé dans un domaine de protection. Un objet *guichet* qui est un point d'entrée du domaine peut permettre d'étendre les droits aux objets internes de l'application sans les rendre toutefois directement accessibles depuis l'extérieur.

Les capacités permettent donc l'expression du contrôle d'accès spécifié dans les objectifs. Nous décrivons maintenant un exemple de système à capacités.

VI.2.2.2 Protection dans le système Hydra

Hydra [Wulf74] est un système développé sur une machine spécifique, dont la philosophie est fondée sur les notions d'objet et de capacité.

Dans Hydra, chaque objet est composé d'une partie contenant des données (*partie−donnée*) et d'une partie contenant des capacités (*C−liste).* Un objet peut être notamment une instance, un type ou une procédure. Un type définit le comportement d'un ensemble d'objets (l'équivalent d'une classe Guide) et il est composé également d'une partie−donnée et d'une *C−liste* qui contient les capacités vers les procédures (méthodes) appelables sur les instances de ce type. Un objet procédure contient le code de la procédure dans sa partie−donnée et les capacités dont il a besoin dans sa *C−liste.*

A l'exécution, une *C−liste* (appelée *LNS* pour *Local Name Space*) est associée à chaque exécution de procédure et correpond à une pile de capacités réalisant les variables locales à la procédure. La *LNS* est commutée à chaque appel de procédure et elle détermine l'espace d'adressage du domaine de protection courant. A l'appel, cette *LNS* est initialisée avec les capacités de la *C−liste* de la procédure appelée et avec les capacités passées en paramètres de l'appel. Un domaine de protection est donc associé à chaque exécution de procédure, c'est à dire à chaque appel de méthode.

Un appel de méthode sur un objet est paramétré par l'indice de la capacité sur l'objet appelé dans la *LNS*, ainsi que par un indice de méthode dans la *C−liste* du type de l'objet appelé.

A la création d'une procédure, il est possible de spécifier que certains paramètres de type capacité doivent voir leurs droits augmentés. Les capacités sur les objets diffusées donnent des droits d'appel de méthode, mais ne donnent pas les droits pour une manipulation directe de leur état. Lorsqu'elles sont passées à une méthode de leur type, les droits de ces capacités sont augmentés pour permettre la manipulation dans la méthode.

La figure Fig. 6.3 illustre un appel de méthode. Avant l'appel de méthode sur l'objet *O2*, *O1* est en cours d'exécution. Le domaine de protection courant a une *LNS* qui contient une capacité sur l'objet courant avec les droits de manipulation directe permettant de lire l'état de l'objet (et en particulier de copier une capacité de la *C−liste* dans la *LNS* afin de l'utiliser pour appeler une méthode), ainsi qu'une capacité sur l'objet *O2* permettant l'appel de méthode. L'appel d'une méthode sur l'objet *O2* provoque un changement de domaine. La

LNS du nouveau domaine contient alors une capacité sur l'objet appelé avec des droits amplifiés, ainsi que des capacités passées en paramètres.

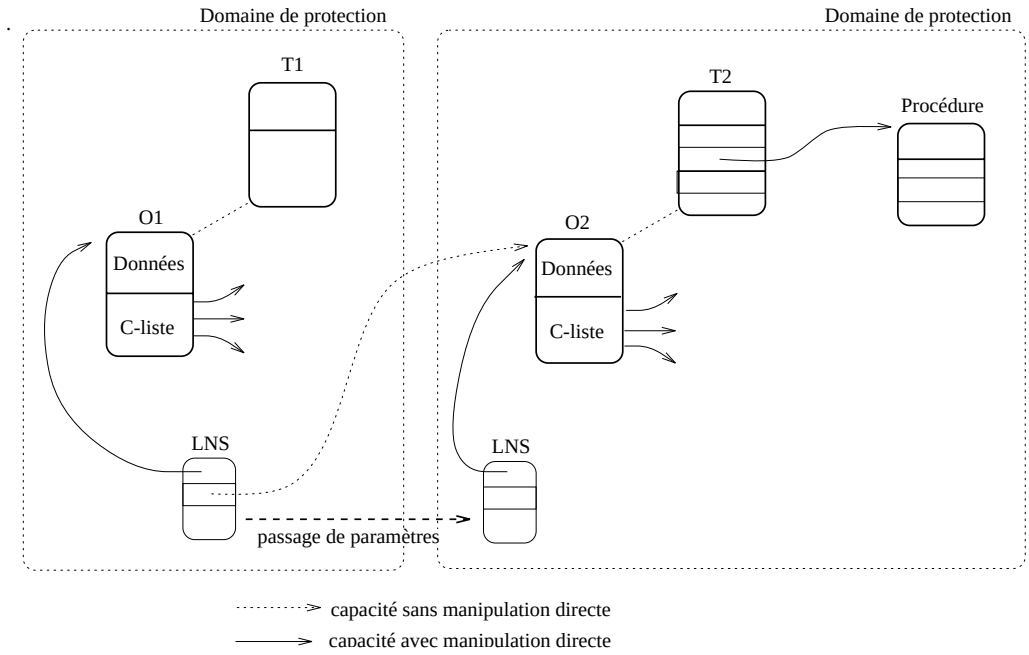

Fig. 6.3 : Protection dans Hydra

VI.2.2.3 Conclusion

A l'origine, les systèmes à capacités ont été introduits pour résoudre les problèmes de méfiance mutuelle présents dans la plupart des systèmes monolithiques, où une partie erronée du système (en mode maître) a tout pouvoir sur le système en entier. Dans Hydra par exemple, une grande partie du système est composée de sous−systèmes protégés et mutuellement méfiants. L'intérêt est de pouvoir modifier une partie du système sans le mettre en péril.

Le problème des systèmes à capacités a longtemps été de n'exister que sur des machines spécifiques, donc chères. Le matériel permettait de contrôler l'allocation et la modification des capacités. Des systèmes à capacités existent aujourd'hui sur des machines banalisées, la protection des capacités étant assurée par encryptage de clés [Mullender86].

VI.3 Contrôle d'accès dans Eliott

Nous présentons maintenant les principes de réalisation du contrôle d'accès dans le noyau Eliott [Hagimont92]. Les mécanismes fournis par le noyau pour le contrôle d'accès aux objets sont fondés sur la définition de listes d'accès. Nous présentons tout d'abord les contraintes qui ont influencé la définition de ces mécanismes. Nous décrivons ensuite la façon dont le modèle a été réalisé en accord avec les principes de conception décrits au chapitre V.

VI.3.1 Contraintes

Pour définir les mécanismes que va offrir notre noyau, il convient d'énumérer les contraintes qui doivent être prises en compte :

- Dans notre environnement d'exécution, seul le premier appel à une méthode est interprété. Cela signifie qu'il est possible que plusieurs appels de méthodes s'exécutent sans que le système en soit averti ; il n'est pas possible de réaliser un contrôle des droits d'accès dans le noyau pour chaque appel de méthode.
- Le noyau Eliott peut supporter des langages non sûrs(1). Il est donc possible que les paramètres transmis depuis le code d'une méthode lors d'un appel à une primitive du système soient erronés. On ne peut donc pas fonder la vérification du contrôle d'accès sur la validité de ces paramètres.
- L'isolation des usagers est primordiale. Le fait qu'un usager du système puisse contourner les mécanismes de protection sur ses propres objets n'est pas très important, mais on veut garantir de façon sûre la validité du contrôle d'accès entre des objets de propriétaires différents.

VI.3.2 Proposition

Notre proposition est présentée en deux parties, la première concernant le contrôle en fonction de listes d'accès simples et la deuxième proposant une extension permettant de faire varier les droits d'accès d'un processus en fonction de son contexte d'exécution.

VI.3.2.1 Contrôle en fonction des listes d'accès

On a vu que les contraintes induites par le mécanisme d'appel de méthode interdisent le contrôle systématique des droits d'accès par le système. Pour conserver notre schéma d'appel interprété au premier appel, nous avons opté pour une solution où l'on met en place, au premier appel de méthode sur un objet, un schéma de liaison dépendant des droits d'accès et limitant les méthodes accessibles.

Pour ce faire, nous définissons la notion de **vue**. Une vue, définie dans une classe, est un ensemble de méthodes autorisées. Un exemple de définition de vue est le suivant. La classe *Fichier* définit les méthodes *lire* et *écrire*. La classe *Fichier* définit les vues :

```
Lecture_écriture  =  (lire,écrire)
Lecture_seule     =  (lire)
Aucun_droit       =  ()
```

Une liste d'accès associée à un objet donne pour chaque usager la vue que cet usager a sur cet objet, ce qui définit un ensemble de méthodes autorisées sur l'objet. Par exemple, si un objet de classe *Fichier* a la liste d'accès suivante :

```
((U1,Lecture_seule)(U2,Lecture_écriture))
```

cela signifie que l'usager *U1* peut seulement appeler *lire* sur cet objet, alors que *U2* peut appeler *lire* et *écrire.*

---

(1) Dont la confiance envers le code généré par le compilateur est limitée.

A l'exécution, le contrôle en fonction des listes d'accès est alors réalisé en associant dans le segment de liaison de la classe une sous−table de vecteurs d'accès à chaque vue définie dans la classe. Chaque vue est représentée dans le segment de liaison de la classe par *N* vecteurs d'accès, *N* étant le nombre de méthodes de la classe. Pour une vue donnée dans le segment de liaison, chaque vecteur d'accès correspond à une méthode et la liaison de la référence à cette méthode n'est autorisée que si la méthode est autorisée par la vue.

Lors de la liaison de la référence d'un objet à sa classe, la liste d'accès de l'objet est consultée pour en extraire la vue associée à l'usager courant et le vecteur d'accès de l'objet à sa classe est mis à jour pour désigner la vue associée aux droits de l'usager. Le deuxième champ de ce vecteur d'accès pointe donc sur la sous−table correspondant à cette vue dans le segment de liaison de la classe.

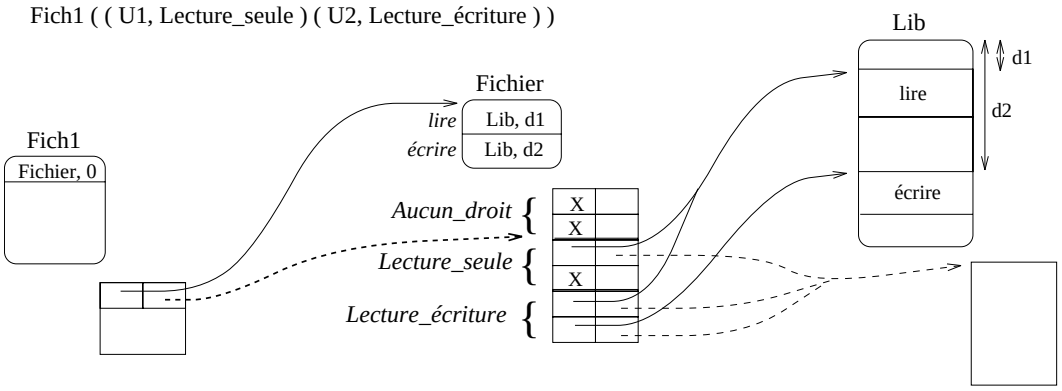

Fig. 6.4 : Réalisation du contrôle par liste d'accès

Cette solution est illustrée sur la figure Fig. 6.4. Sur cette figure, un objet *Fich1* de classe *Fichier* est couplé dans une tâche dans un Domaine s'exécutant pour le compte de l'usager *U1*. La liste d'accès de l'objet *Fich1* spécifie que *U1* doit utiliser *Fich1* avec la vue *Lecture_seule*. La liaison de la référence de *Fich1* vers sa classe désigne dans le segment de liaison de la classe la partie correspondant à la vue *Lecture_seule*. Dans cette vue, une tentative de liaison du deuxième vecteur d'accès retournera une erreur.

Le code généré pour un appel de méthode est le même qu'auparavant. La protection n'est testée que lors de la liaison des références de l'objet vers sa classe et de la classe vers les méthodes. La liaison d'une référence de la classe vers une méthode doit consulter la définition des vues stockée dans la classe et ne lier cette référence que si la vue l'autorise.

Notons que la liaison de la référence entre l'objet et sa classe dépend de l'usager associé au Domaine Guide courant et non du propriétaire d'objets associé à la tâche courante.

Cette structure empêche la prise en compte immédiate d'une modification d'une liste d'accès. Elle ne sera prise en compte qu'à la prochaine liaison.

Ce schéma de liaison étant mis en place dans l'espace utilisateur, on peut se poser la question de la résistance à la fraude du mécanisme de protection. L'isolation des usagers repose sur le fait que des objets de propriétaires différents sont couplés dans des tâches différentes. Le schéma de liaison qui contrôle l'accès aux méthodes est mis en place dans une tâche associée au propriétaire de l'objet appelé. L'exécution d'une méthode sur un objet ne pourra pas aller modifier ce schéma de liaison si l'objet appelé appartient à un autre propriétaire, car l'objet appelé sera couplé dans une tâche différente. Un usager peut toutefois écrire des programmes qui vont modifier les schémas de liaison et autoriser l'exécution de méthodes interdites sur ses propres objets, de même qu'il peut modifier directement par ses programmes l'état de ses objets couplés dans la tâche courante. Si un usager écrit un programme de ce type et "instancie" la classe correspondante, il ne pourra frauder que la protection des objets qui lui appartiennent, ce qui n'a que peu d'intérêt. La seule façon de "pirater" les objets d'un autre usager serait de faire en sorte que cet autre usager instancie lui même la classe permettant la fraude. L'utilisation (instanciation) d'une classe signifie donc une confiance envers le code de cette classe.

VI.3.2.2 Contrôle en fonction du contexte d'un processus

Une solution à ce problème a été proposée dans le projet Birlix [Kowalski90] en offrant la possibilité d'inclure dans les listes d'accès le contrôle de la classe appelante. On peut donc gérer une liste d'accès composée de triplets (*U, C, M*) signifiant que l'appel par l'usager *U* depuis la classe *C* de la méthode *M* est autorisé.

Si on reprend l'exemple du client−service énoncé en section VI.1, il suffit de spécifier pour les objets internes du service que seuls les appels provenant de la classe du *guichet* sont autorisés.

Cette solution n'est pas réalisable de façon sûre dans notre cas pour deux raisons :

- Etant donné que le noyau Eliott peut supporter des langages non sûrs, l'identité de la classe appelante, si elle est transmise par du code s'exécutant en mode utilisateur, peut avoir été falsifiée.
- Il n'est pas possible de garantir la connaissance de la classe appelante dans le noyau, puisque les appels de méthode ne sont interprétés qu'au premier appel (ne passent pas tous par le noyau).

Nous avons donc opté pour une solution permettant de protéger de façon sûre les applications, fondée sur la séparation entre les domaines de protection des tâches. Lorsqu'un appel de méthode implique des objets de propriétaires différents, cet appel est réalisé par appel de procédure à distance entre deux tâches et il est interprété. De plus, lorsque la tâche contenant l'objet appelé reçoit un appel, la seule donnée qu'elle peut identifier de façon sûre est l'identité de la tâche appelante, c'est à dire le propriétaire d'objet associé statiquement à cette tâche. Elle peut donc connaître le propriétaire de l'objet appelant.

Le test de la classe appelante correspond à une vérification sur un attribut de l'objet appelant qui est sa classe. Le but est de discriminer les appels en fonction de l'objet appelant. Le principe de notre modèle de protection est d'utiliser un autre attribut de l'objet appelant qui est son propriétaire, toujours pour discriminer les appels.

Ajouter dans les listes d'accès le propriétaire de l'objet appelant nous parait trop général pour le type de problèmes à résoudre (bien que cette solution soit techniquement réalisable). Nous proposons donc, pour un objet, de discriminer les appels provenant d'objets du même propriétaire des appels provenant d'objets appartenant à des propriétaires différents.

A chaque objet est attaché un *attribut de visibilité*, qui selon sa valeur (visible, invisible) indique si l'objet appartenant à un propriétaire *P* peut ou non être appelé depuis un objet appartenant à un autre propriétaire que *P*. Par défaut, cet attribut est positionné à la valeur "invisible". Cela permet de résoudre le problème de la variation des droits d'un processus comme nous le montrons sur l'exemple du client−service. L'administrateur du sous−système réalisant ce service crée un ou plusieurs objets *guichets* dont l'attribut est "visible" et les classes réalisant ce service indiquent que tous les objets internes ont un attribut "invisible". Ces objets appartiennent tous à l'administrateur de ce sous−système et seuls les objets *guichets* sont accessibles directement depuis des objets appartenant à d'autres propriétaires.

Le principe de ce mécanisme de protection consiste donc à définir les points d'entrée de l'application qu'elle rend visibles aux utilisateurs. Ce contrôle est bien moins fin que celui qui aurait consisté à contrôler la classe de l'objet appelant, mais notre mise en œuvre ne nous permet pas de la vérifier de façon sûre. Cette solution nous semble un bon compromis dans la mesure où elle permet de résoudre la plupart des problèmes sans introduire de faille dans le système de protection.

Elle peut être comparée à la solution de Multics dans le sens où elle consiste également à définir un sous−système s'exécutant dans un contexte protégé avec des points d'entrée définis. Toutefois, le problème de méfiance mutuelle est dans notre cas résolu, puisque l'appel d'un point d'entrée du sous−système ne change pas les droits du sous−système sur son client.

L'attribut de visibilité est un booléen associé à toute instance. Comme tous les objets couplés dans la même tâche appartiennent au même propriétaire, l'attribut n'a besoin d'être testé que lors des appels entre tâches. Il faut toutefois remarquer qu'un appel entre tâches n'est pas forcément un appel entre des objets de propriétaires différents, puisqu'un Domaine peut inclure deux tâches associées au même propriétaire d'objets sur des sites différents. La technique utilisée pour identifier la tâche contenant l'objet appelant (donc le propriétaire de l'objet appelant) est décrite plus en détail dans le chapitre suivant.

VI.4 Conclusion

Ce chapitre a présenté les principes de mise en œuvre du contrôle d'accès dans le noyau Eliott.

Les mécanismes permettant le contrôle d'accès sont fondés sur trois notions :

- Les vues sont des filtres définis dans les classes qui déterminent un ensemble de méthodes autorisées.
- Les listes d'accès associées aux objets permettent d'attribuer une vue à chaque usager du système.

- L’attribut de visibilité associé à chaque objet permet la construction de sous−systèmes protégés.

Ces mécanismes permettent de résoudre les problèmes classiques de protection sans reposer sur la sûreté des langages supportés. La sûreté de ces mécanismes de protection repose principalement sur la séparation entre les domaines de protection des tâches, utilisés pour l’isolation des objets par propriétaires d’objets.

Chapitre VII

Réalisation du noyau Eliott

Le but de ce chapitre est de présenter de façon détaillée la réalisation du noyau Eliott sur le micro−noyau Mach 3.0.

Après une description de l'architecture logicielle choisie pour la réalisation, nous présentons brièvement les caractéristiques de Mach utiles à la compréhension. Puis, la réalisation des différents composants est décrite, en donnant une attention plus particulière à l'adressage des objets, qui fournit le modèle à objets primitif et à la gestion de la protection dans ce modèle, car ils constituent les aspects originaux de mon travail de thèse.

VII.1 Architecture générale

La figure Fig. 7.1 montre une vue globale du système.

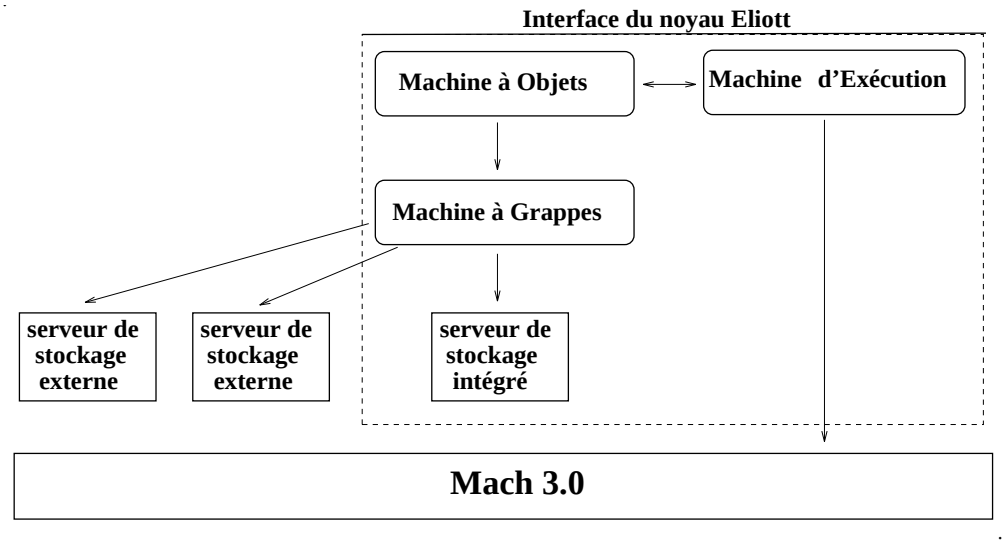

Fig. 7.1 : Architecture globale du noyau Eliott

Le noyau Eliott fournit l'interface utilisée par les réalisateurs de compilateurs. Cette interface donne accès à des services relatifs à la gestion de l'exécution (comme la création d'Activité dans un Domaine Guide) et à des services relatifs à la gestion des objets (comme la création ou l'appel d'objets).

Nous appelons *machine à objets* le composant du système qui offre la gestion d'objets ; les entités gérées sont les instances, les classes et les librairies de code. Cette

machine fournit donc d'une part une interface permettant aux compilateurs de créer des classes et des librairies contenant le code des méthodes compilées, mais également une interface utilisée à l'exécution pour la création des instances et les appels à ces instances. Cette machine est décrite en section VII.4. La machine à objets est construite en utilisant la notion de segment décrite au chapitre précédent. Cette notion permet, grâce à un adressage segmenté, le partage d'objets couplés à des adresses différentes dans différentes tâches et la liaison dynamique des références désignant les objets.

La machine à objets est construite sur la *machine à grappes* présentée en section VII.3. La grappe est l'unité de partage entre les tâches. La machine à grappes fournit essentiellement aux niveaux supérieurs les mécanismes de couplage et de recopie atomique des grappes en mémoire de stockage. Le stockage est assuré par des serveurs de stockage. Un serveur de stockage rapide est intégré au système, mais il est possible d'utiliser des serveurs externes assurant un service de meilleur qualité (haute disponibilité par exemple).

Enfin, la *machine d'exécution* met en œuvre les Domaines multi−tâches et les Activités. Elle fournit également le mécanisme d'extension, permettant aux activités de changer de tâche d'exécution.

VII.2 Quelques caractéristiques de Mach 3.0

Le but de cette section est de décrire les caractéristiques de Mach 3.0 qui sont essentielles pour la réalisation de notre noyau.

Cette description est composée de trois parties :

- Les structures d'exécution : les *tâches* et les *threads*(1).
- Les communications fondées sur la notion de *port*.
- La gestion de la mémoire virtuelle à l'aide de *paginateurs externes*.

VII.2.1 Les tâches et les processus légers

La tâche est l'unité de structuration du système. C'est un environnement d'exécution pour les processus légers. Elle comprend un espace virtuel d'adressage privé et un ensemble de droits d'accès à des ressources du système (par exemple les ports). Une tâche est locale à un site.

Le processus léger est l'unité de base d'allocation du processeur. Un processus léger s'exécute dans l'espace virtuel d'une tâche et tous les processus légers d'une tâche partagent ses ressources.

La notion de processus dans un système Unix est équivalente à une tâche avec un unique processus léger.

Les tâches et les processus légers sont désignés par des ports. La connaissance du port identifiant une tâche ou un processus léger permet d'appeler des primitives de Mach pour modifier son état.

---

(1) que nous appellerons dans la suite "processus légers".

VII.2.2 Les communications

Les communications dans Mach sont fondées sur les notions de port et de message.

Un message est une suite typée de données adressées à un port.

Un port est une boîte aux lettres à laquelle les messages peuvent être envoyés, c'est à dire une file de messages. A un instant donné, une et une seule tâche possède le droit de lecture sur un port, les autres tâches peuvent obtenir des droits d'émission de messages sur ce port. La tâche qui envoie un message sur un port peut s'exécuter sur un site différent de celui de la tâche qui possède le droit de lecture sur ce port.

Les ports sont aussi utilisés pour désigner les ressources dans Mach (les tâches et les processus légers notamment).

Les ports dans Mach sont protégés, car la transmission de ports dans un message entre des tâches est contrôlée par le noyau. Un port est désigné dans une tâche ayant des droits sur ce port par un nom local à cette tâche. Lorsque ce nom local est utilisé dans un message pour transmettre des droits sur ce port à une autre tâche, ce nom local est traduit par le noyau et la tâche réceptrice désigne ce port par un autre nom local privé alloué par le noyau. Le noyau Mach gère pour chaque tâche la correspondance entre ses noms locaux et les ports. Ainsi, une tâche ne peut pas se forger des droits sur un port. De même, il n'est pas possible de partager un nom de port entre deux tâches dans de la mémoire partagée. Une tâche ne peut avoir que les droits qui lui ont été explicitement transmis.

Une vision globale de deux machines utilisant Mach est donnée sur la figure Fig. 7.2.

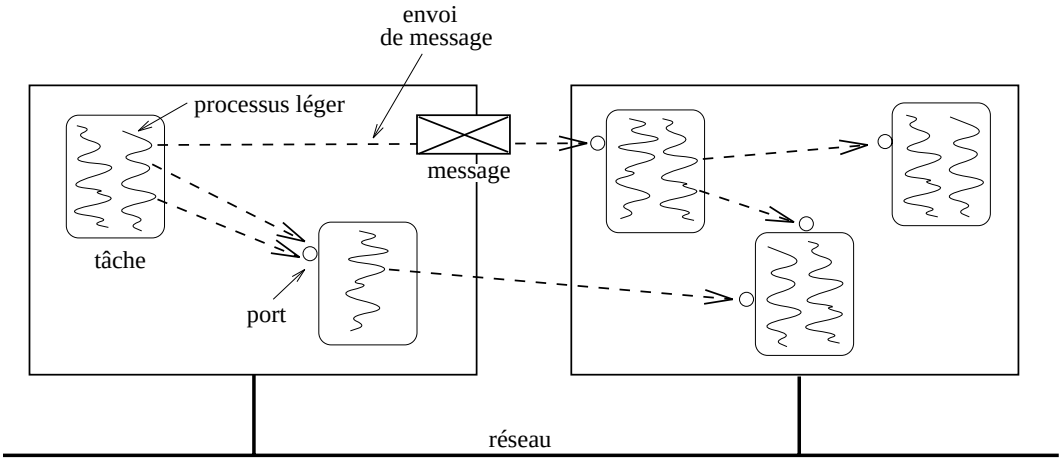

Fig. 7.2 : Vue globale des abstractions de Mach

VII.2.3 La gestion de la mémoire virtuelle

L'espace virtuel d'une tâche est constitué d'un ensemble de régions. Une région est une fenêtre d'adressage composée d'un ensemble de pages consécutives.

Le va et vient en mémoire centrale des pages d'une région peut être géré par le noyau Mach ou par un gestionnaire de pagination externe au noyau appelé paginateur externe.

Il y a deux façons de créer une région :

- Soit par allocation explicite d'une région dans l'espace virtuel d'une tâche. Sa pagination est alors confiée au gestionnaire de pagination de mémoire virtuelle (appelé *default pager*) présent dans le noyau Mach.
- Soit par couplage d'un segment partagé géré par un paginateur externe. Un paginateur est une tâche Mach s'exécutant hors du noyau.

  Un segment partagé est créé par le paginateur et il est désigné par un port appartenant au paginateur. Lorsque le couplage d'un segment dans une tâche est demandé, le port désignant ce segment doit être spécifié. Une région est alors allouée dans l'espace virtuel de la tâche. L'adresse virtuelle de cette région peut être spécifiée lors de la demande de couplage, ou allouée par le noyau Mach dans une zone libre de l'espace virtuel de la tâche.

  A chaque faute de page dans cette région, un message est envoyé par le noyau au paginateur sur le port identifiant le segment couplé. Le paginateur est alors chargé de répondre à cette faute de page en retournant la page au noyau. De même, si cette page doit être vidée de la mémoire centrale, elle est retournée par le noyau au paginateur. Ce principe est illustré sur la figure Fig. 7.3.

  Plusieurs tâches s'exécutant sur des sites différents peuvent coupler le même segment partagé. Le paginateur peut alors recevoir des fautes de pages provenant de plusieurs noyaux Mach. Il doit alors assurer la cohérence des données partagées, en limitant le nombre de copies de pages et les droits sur ces copies (lecture/écriture), donnés aux noyaux. Pour assurer la disponibilité des pages, le paginateur a la possibilité de réclamer une copie de page à un noyau, ou de demander la restriction de ses droits.

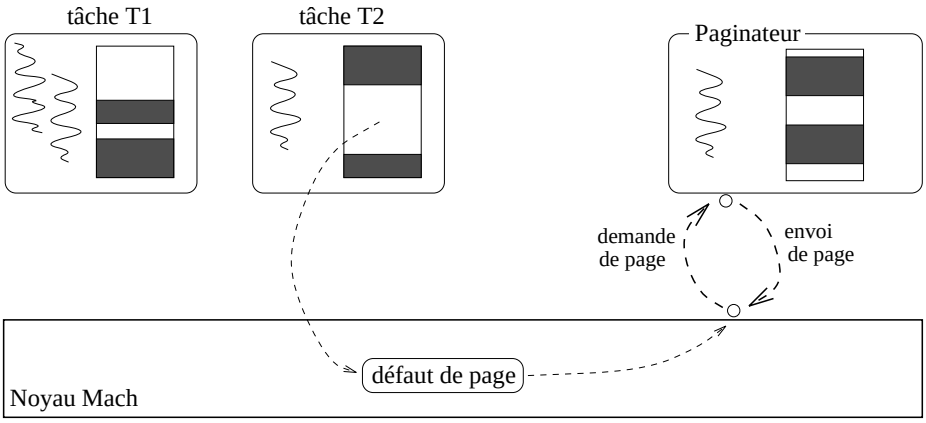

Fig. 7.3 : Les paginateurs de Mach

VII.3 La machine à grappes

La fonction essentielle de la machine à grappes est de permettre la mise en œuvre du partage et de la persistance des grappes. Elle est donc séparée en deux sous−machines correspondant respectivement à la gestion des grappes en mémoire d'exécution et à la gestion des grappes en mémoire de stockage. L'organisation de ces sous−machines est décrite ci−dessous.

VII.3.1 Gestion de grappes en mémoire d'exécution

Organisation générale

La gestion des grappes en mémoire d'exécution est mise en œuvre à l'aide de serveurs appelés gérants de mémoire qui sont des paginateurs pour le micro−noyau Mach 3.0. Il y a en principe un gérant de mémoire par site, bien qu'un site puisse fonctionner sans gérant de mémoire (avec le gérant de mémoire d'une autre machine).

Un gérant de mémoire est chargé de la gestion d'un ensemble de volumes, donc des grappes contenues dans ces volumes. Un volume est géré par un gérant de mémoire unique. On dit alors que ce volume est *monté* sur le site d'exécution de ce gérant de mémoire. L'opération de montage de volume est une opération d'administration du système. La description des montages de volumes est stockée dans des tables dupliquées pour partage par les gérants de mémoire.

Un gérant de mémoire permet le couplage des grappes dont il a la charge. Une identification de grappe est constituée des deux premières parties d'un identificateur d'objet. Elle est donc constituée d'un identificateur de volume et d'un identificateur de grappe local à ce volume (Fig. 7.4).

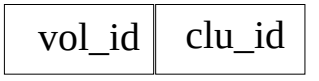

Fig. 7.4 : Format d'un identificateur de grappe

Les grappes sont couplées dans les tâches à la demande par la machine à objets. Une requête de couplage de grappe est envoyée par message au gérant de mémoire du site sur lequel est monté le volume qui la contient (ce volume est donné dans l'identificateur de la grappe et une grappe ne change jamais de volume). La grappe est alors couplée à une adresse quelconque choisie par Mach.

Le gérant mémoire est alors responsable de la pagination de la grappe. Il répond donc aux fautes de page sur cette grappe. Les données de la grappe sont lues en mémoire de stockage et données au noyau Mach. Lorsque le noyau Mach envoie au gérant de mémoire une page pour faire de la place en mémoire centrale, cette page est stockée dans une zone temporaire (*partition de pagination*), ce qui permet de garder la grappe dans un état cohérent pendant l'exécution (l'état initial avant couplage), jusqu'au découplage de la grappe.

Protection

Le couplage d'une grappe n'est pas forcément autorisé. En effet, une grappe appartient à un propriétaire d'objets (tous les objets d'une grappe appartiennent au même propriétaire) et une tâche ne peut coupler que des grappes appartenant au propriétaire associé à cette tâche. Le gérant de mémoire qui reçoit une demande de couplage doit donc vérifier si la tâche d'où provient la demande est autorisée à coupler cette grappe. Si le couplage est interdit, la machine à grappes retourne l'identité du propriétaire de la grappe. La machine à objets peut alors appeler la machine d'exécution pour changer de tâche d'exécution et aller dans une tâche associée au propriétaire et autorisant le couplage.

Pour que cette protection soit sûre, il faut être capable d'authentifier la tâche demandant le couplage. La demande de couplage, qui est réalisée par envoi de message au gérant de mémoire, contient le port Mach qui identifie la tâche émettrice. Le gérant de mémoire, qui est une tâche privilégiée de notre système, doit donc avoir accès à l'association entre le port identifiant une tâche et l'usager (propriétaire) associé à la tâche, afin de tester si le couplage est autorisé. Cette association est gérée par la machine d'exécution. Une tâche ne peut donc pas se faire passer pour une autre, à condition que les tâches n'échangent pas leur port d'identification.

Remarquons qu'il est également possible de définir des grappes dont le couplage n'est autorisé que sur un seul site à la fois. On force ainsi toutes les activités à partager cette grappe sur le même site. Ce couplage peut aussi être interdit si la tâche qui le demande se trouve sur un site incompatible (hétérogène) par rapport au type de machine associé à la grappe. Dans ces deux cas, la machine à grappes retourne l'identification d'un site sur lequel la grappe peut être couplée. La machine à objets peut alors demander un changement de tâche d'exécution.

Modes de couplage et types de grappe

On distingue tout d'abord deux modes de couplage de grappes. Les grappes peuvent être couplées en lecture−seule ou en lecture−écriture. On distingue ensuite différents types de grappes et chaque type de grappes a un mode de couplage par défaut :

- Les grappes destinées à contenir du code et des classes.

  Le mode de couplage par défaut est lecture−seule. Les grappes peuvent alors être couplées dans toutes les tâches (quel que soit le propriétaire associé à la tâche) pour permettre l'exécution du code des méthodes sur les instances. Le gérant de mémoire peut alors donner des copies en lecture des pages de la grappe, sur tous les sites où elle est couplée.

  Le propriétaire d'une telle grappe peut toutefois demander le couplage en écriture pour recompiler une classe.

- Les grappes destinées à contenir des instances.

  Ces grappes sont toujours couplées en lecture−écriture et elles ne peuvent être couplées que dans une tâche associée au propriétaire de la grappe.

Possibilité de modifier la taille des grappes

La machine à grappes permet d'agrandir une grappe déjà couplée dans plusieurs tâches. Etant donné que Mach ne permet pas d'agrandir une zone d'espace virtuel dans laquelle une

grappe est couplée, nous gérons une grappe couplée dans une tâche comme une suite de *blocs* de grappe couplés dans des zones non contiguës de l'espace virtuel de la tâche. Dans une tâche, il est possible de demander le couplage de tous les blocs qui ne sont pas encore couplés, ou de demander la création d'un nouveau bloc.

Interface

L'interface de la machine à grappes en mémoire d'exécution est composée des fonctions suivantes :

- **clu_CreateMap**

  Permet la création d'une grappe. La grappe est alors couplée dans la tâche courante et appartient au propriétaire associé à cette tâche.

- **clu_Map**

  Permet le couplage d'une grappe. La grappe est couplée (si le couplage est autorisé) à une adresse choisie par Mach dans l'espace virtuel de la tâche courante.

- **clu_Unmap**

  Permet d'annuler le couplage d'une grappe dans la tâche courante. Cette fonction est appelée dans chaque tâche à la fin de l'exécution du Domaine pour le compte duquel la tâche s'exécute.

- Des primitives de gestion de blocs de grappe.

VII.3.2 Gestion de grappes en mémoire de stockage

Organisation générale

La gestion de l'espace de stockage gère une mémoire constituée d'un ensemble de volumes stockés sur les disques des stations de travail.

Elle fournit à l'administrateur du système des primitives de création, destruction et de changement de taille des volumes. Chaque volume est identifié par un numéro de volume global au système. C'est l'identificateur de volume qui se trouve dans le premier champ d'une *SysRef.*

Elle fournit à la gestion des grappes en mémoire d'exécution (aux gérants de mémoire) des primitives de gestion des grappes dans les volumes. Un volume monté sur un site doit être accessible au gérant de mémoire de ce site par des opérations de lecture et d'écriture.

Un service de stockage gérant des volumes peut être réalisé par une librairie ou par un serveur. Si ce service est directement réalisé par une librairie, cette librairie gère un disque local au site du gérant de mémoire. Si ce service est réalisé par un serveur, une librairie d'envoi de message à ce serveur fournit l'interface d'accès au serveur. Dans les deux cas, la librairie est liée au gérant de mémoire.

Interface

L'interface de la machine à grappes en mémoire de stockage est composée des fonctions suivantes :

- Les fonctions de gestion des volumes : **vol_Create**, **vol_Resize**, **vol_Delete**. Ces fonctions permettent respectivement de créer, ajuster la taille et détruire les volumes.
- Les fonctions de gestion des montages de volumes : **vol_Mount**, **vol_Unmount**. Elles permettent le montage et l'annulation du montage d'un volume sur un site.
- Les fonctions de gestion des grappes dans les volumes : **vol_CluCreate**, **vol_CluGetDesc**, **vol_CluDestroy**, **vol_CluResize**, **vol_CluRead**, **vol_CluWrite**. Elles permettent notamment la création et la destruction des grappes dans les volumes et la lecture et l'écriture des données stockées dans ces grappes.

Une vue globale de la machine à grappes est donnée sur la figure Fig. 7.5.

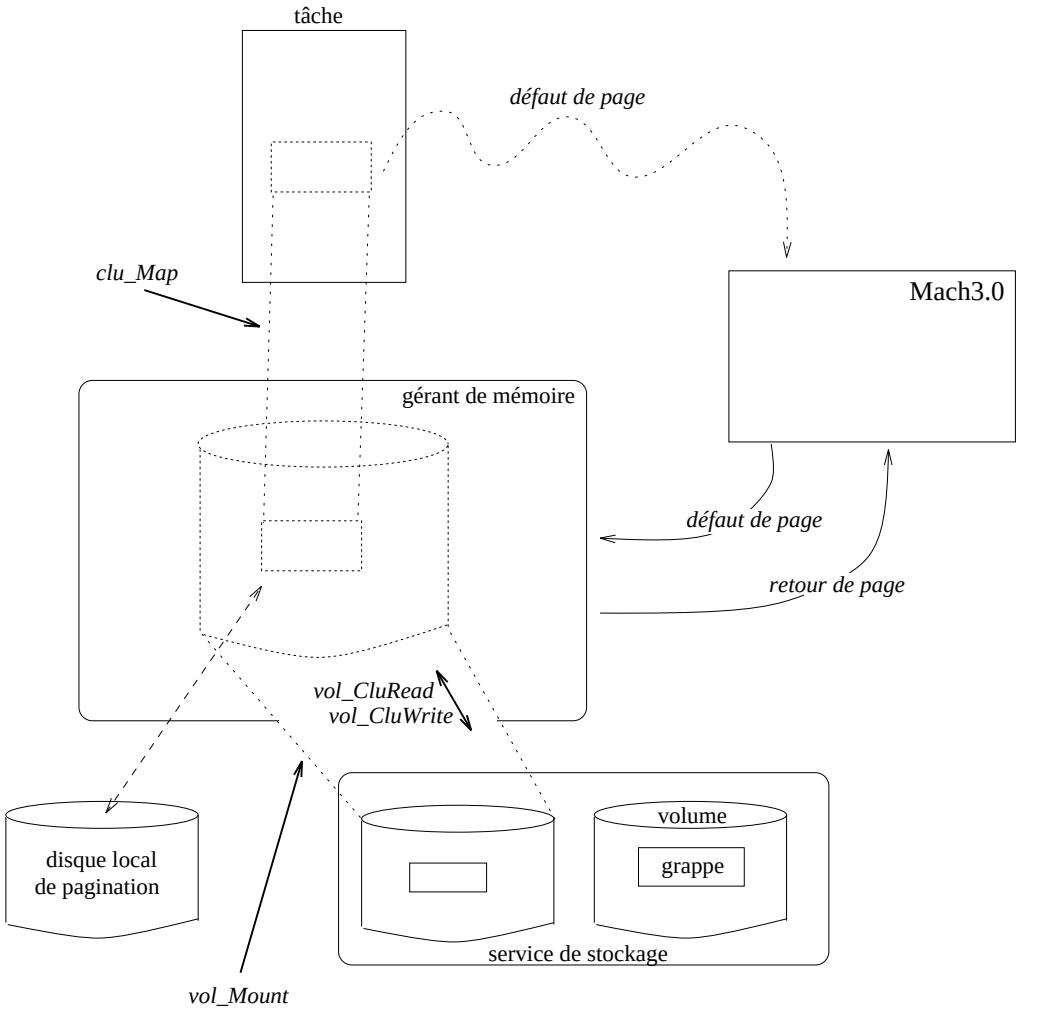

Fig. 7.5 : Architecture de la machine à grappes

VII.4 La machine à objets

Le but de la machine à objets est de fournir le modèle à objets primitif décrit en IV. Les abstractions gérées dans ce modèle, à savoir les instances, les classes et les librairies de code, sont réalisées par des segments. La machine à objets est donc composée de deux parties :

- La première fournit la gestion des segments dans les grappes, ainsi que les mécanismes de base pour l'adressage de ces segments. Nous présenterons donc respectivement la gestion des segments dans les grappes, la localisation des segments, la structure des segments et la liaison des références aux segments.
- La deuxième partie enrichit la première en associant un type à chaque segment. Un segment est alors un segment d'instance, un segment de classe ou un segment de code. Nous détaillerons la structure de ces segments typés et leur utilisation par l'environnement d'exécution, principalement pour réaliser des appels de méthode.

VII.4.1 Gestion des segments

Gestion des segments dans une grappe

Un segment dans une grappe est une zone contiguë de données. L'allocation des segments dans une grappe est faite linéairement. Les destructions de segments peuvent fragmenter l'espace de la grappe. Lorsqu'un seuil fixé de fragmentation est atteint dans la grappe, celle−ci est restructurée (comprimée) lorsqu'elle est découplée.

Dans chaque grappe, une table, indicée en fonction de la référence du segment, permet de retrouver l'état du segment, ainsi qu'une zone contenant les informations de protection associée à ce segment et servant à la gestion des listes d'accès pour les instances et des vues pour les classes.

Chaque bloc de grappe couplé est géré comme une grappe et la recherche d'un segment consiste à chercher dans tous les blocs de la grappe. Une opération permet de restructurer une grappe pour quelle puisse être gérée en un bloc unique.

Localisation des segments

Il est nécessaire de connaître dans chaque tâche quelles sont les grappes couplées dans la tâche ainsi que leur adresse de couplage dans l'espace virtuel de la tâche. Une table appelée *cache de grappes* est donc gérée dans cet espace virtuel et enregistre chacun de ces couplages.

Comme on l'a vu au chapitre V, un catalogue de migration est géré dans chaque grappe et enregistre les segments déplacés arrivés dans la grappe. Lorsque le couplage d'une grappe est réalisé par la machine à grappes, ce catalogue est chaîné dans une liste globale des catalogues d'objets déplacés.

Lorsqu'une référence à un segment doit être liée dans une tâche (ce qui nécessite l'obtention de l'adresse de couplage du segment), le segment est recherché dans sa grappe de création, donnée par la *SysRef* du segment, et les liens de poursuite sont suivis si le segment a été déplacé, tant que les grappes visitées sont dans le cache des grappes couplées. Si une

grappe doit être couplée, le catalogue des segments déplacés de la tâche est utilisé et la grappe n'est couplée que si le segment n'est pas dans ce catalogue.

Structure d'un segment

L'état d'un segment contient une partie propre et des données de liaison. Les données de liaison sont constituées de deux tables : la table des définitions externes (TDE) contient les déplacements des différents éléments du segment qui sont adressables depuis d'autres segments, la table des références externes (TRE) contient tout ou partie des références à d'autres segments incluses dans le segment.

La TDE définit sous forme de couples (nom_externe, déplacement) les déplacements dans le segment courant des différents éléments du segment qui sont adressables depuis d'autres segments. Cette table permet la résolution de références de la forme (*SysRef, nom_externe*) en (*SysRef, déplacement*) des différents éléments exportés par le segment. Il est ainsi possible de modifier l'état d'un segment sans avoir à répercuter ces modifications dans les segments référençant ce segment.

La TRE n'a pas à définir l'ensemble des références externes du segment, puisque contrairement au système Multics où les opérations de couplage et de liaison sont implicites, sur notre machine ces opérations sont explicitement demandées au système qui reçoit les informations nécessaires en paramètre. Dans notre cas, la TRE permet de demander une liaison implicite de certaines références, qui seront liées lors de l'allocation du segment de liaison de ce segment. Ces références externes ont la forme (*SysRef, nom_externe*) dans la TRE. Un attribut du segment détermine le nombre *n* d'entrées de la TRE à prélier. Les vecteurs d'accès associés à ces *n* premières entrées sont les *n* premiers dans le segment de liaison.

Liaison d'une référence

Dans chaque tâche, une table appelée *cache de segments* enregistre toutes les liaisons de segments résolues. Ce cache de segments donne principalement l'adresse de couplage du segment et l'adresse de son segment de liaison. La liaison d'une référence à un segment a pour but de mettre à jour un vecteur d'accès en fonction d'une référence (*SysRef, nom_externe*). Elle consulte ce cache et met à jour le vecteur d'accès si le segment est dans le cache, en y ajoutant le déplacement associé à *nom_externe* dans la TDE du segment référencé.

Si le segment n'est pas dans le cache, la liaison de cette référence nécessite la localisation du segment. Dans cette phase de localisation, plusieurs grappes peuvent être couplées. Lorsque l'adresse de couplage du segment est obtenue, un segment de liaison est alloué dans l'espace virtuel de la tâche. Le nombre de vecteurs d'accès dans ce segment de liaison est un attribut du segment. Les entrées de la TRE de ce segment qui doivent être préliées le sont alors, avant que soit mis à jour le vecteur d'accès associé à la référence.

Le processus de liaison d'une référence à un segment est illustré sur la figure Fig. 7.6. Le segment *S* contient une référence externe vers l'entrée *entrée2* du segment *S1*, qui n'est pas liée dans la partie *a)* et qui est liée sur la partie *b)*. Le vecteur d'accès associé à cette référence externe est à l'indice *i(S1)* dans le segment de liaison de *S*. Le segment *S1* exporte deux éléments *début_S1* et *entrée2*, qui sont enregistrés dans sa *TDE*. La *TRE* de

S1 contient une référence externe à l'élément *début_S2* du segment *S2*, qui doit être préliée. Cette référence externe, qui est la première dans la *TRE* de *S1*, se voit associer le premier vecteur d'accès dans le segment de liaison de *S1*. Lorsque la référence externe de *S1* à *S2* est liée, le vecteur d'accès associé à cette référence externe (à l'indice *i(S1)*) désigne le point d'entrée *entrée2* dans le segment *S1* et le segment de liaison de *S1*. La première référence externe dans la *TRE* du segment *S1* a été préliée. Le premier vecteur d'accès dans le segment de liaison de *S1* désigne donc l'élément exporté *début_S2* du segment *S2*.

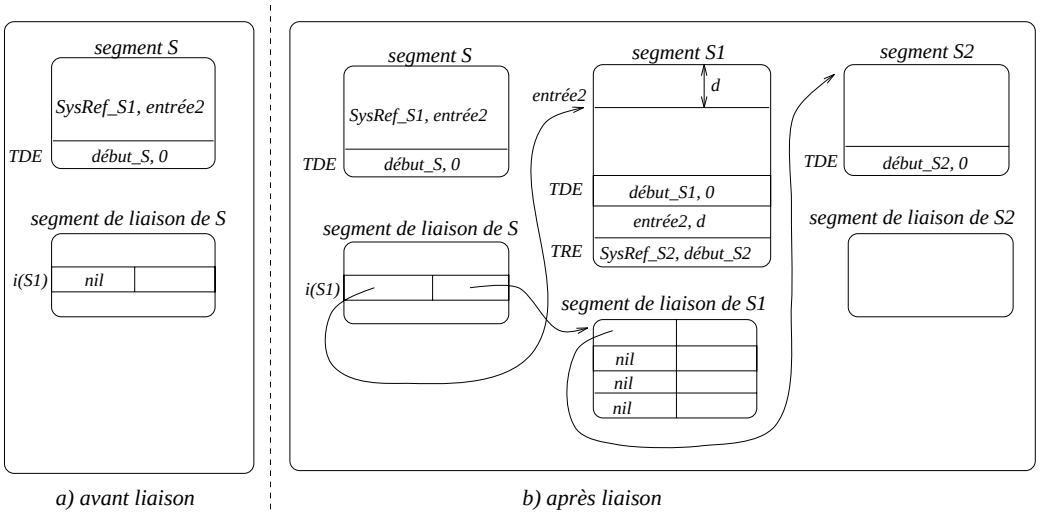

Fig. 7.6 : Processus de liaison d'une référence à un segment

VII.4.2 Gestion des objets

Librairies de code

Une librairie de code est un segment à points d'entrée multiples, destiné à contenir le code des méthodes ou des procédures appartenant aux mêmes classes ou à des classes différentes. Les librairies sont stockées dans des grappes partageables en lecture.

Les points d'entrée d'une librairie de code sont enregistrés dans la TDE du segment. La TRE de ce segment peut être utilisée pour prélier des références à des procédures stockées dans d'autres librairies. Il est également possible de lier dynamiquement ces références externes.

Classes

Une classe est mise en œuvre par un segment qui contient la *SysRef* d'un segment qui sera utilisé comme modèle lors de la création d'une instance de la classe. Ce segment est appelé le modèle d'instance de la classe et l'instanciation est réalisée par copie de ce modèle d'instance. La TRE de la classe contient les références aux méthodes constituant l'interface de la classe. La TDE ne contient qu'une entrée.

La zone de protection du segment réalisant la classe contient l'ensemble des vues définies pour la classe. Le segment de liaison d'un segment de classe contient une table de vecteurs

d'accès (un par méthode) pour chaque vue définie par la classe. Lorsqu'une référence à une méthode doit être liée en utilisant un vecteur d'accès dans une table correspondant à une vue donnée, la définition de la vue est consultée dans la zone de protection du segment de classe et la liaison n'est réalisée qui si la vue autorise l'appel de cette méthode.

Objets

Un objet, instance d'une classe, est mis en œuvre par un segment. La première entrée de la TRE de ce segment contient toujours la *SysRef* du segment qui contient sa classe. De plus, le mécanisme de préliaison est utilisé pour coupler et lier la classe en même temps que l'objet. La TDE ne contient qu'une entrée.

La zone de protection de ce segment contient la liste d'accès à l'objet et définit l'ensemble des méthodes appelables sur cet objet pour chaque usager du système. Lorsqu'un objet est lié dans un Domaine s'exécutant pour le compte d'un usager *U,* la zone de protection du segment instance contenant la liste d'accès à l'objet est consultée et le premier vecteur d'accès du segment de liaison de ce segment (correspondant à la référence à sa classe) est mis à jour pour désigner la table, dans le segment de liaison de la classe, correspondant à la vue associée à *U.*

Appel de méthode

Deux problèmes doivent être résolus pour réaliser l'appel de méthode :

- La gestion des registres Rbo, Rblo et Rblc.

  Etant donné que ces registres sont privés à chaque Activité, ils sont alloués sur les piles de ces Activités. Un appel de méthode est réalisé par un appel de procédure dont les premières instructions consistent à déclarer ces registres comme des variables locales à la procédure et à les initialiser pour qu'ils désignent le bon objet et les bons segments de liaison.

- Les tests de liaison de l'objet appelé et du code de la méthode appelée.

  Un objet appelé est désigné par une *SysRef*. Cette *SysRef* peut être stockée dans l'état d'un objet ou sur la pile (variable d'état ou temporaire). Dans le premier cas, un vecteur d'accès est associé à cette référence dans le segment de liaison de l'objet appelant. Dans le second cas, un vecteur d'accès est déclaré sur la pile et associé à cette variable temporaire.

  Dans les deux cas, lors de l'appel de méthode, il faut tester si le vecteur d'accès associé à la variable est à jour, c'est à dire si le vecteur d'accès désigne d'une part un objet et d'autre part s'il s'agit du bon objet, la variable ayant pu être ré−affectée après liaison. Pour réaliser ce test, chaque segment réalisant une instance contient en entête sa propre *SysRef*, ce qui permet de détecter si un vecteur d'accès désigne bien l'objet désiré. Si le vecteur d'accès n'est pas à jour ou s'il contient une information obsolète, une primitive du système est appelée pour effectuer la liaison de la référence.

  La liaison de la référence de l'objet appelé à sa classe n'a pas besoin d'être testée, puisque cette référence, qui se trouve dans la TRE du segment réalisant l'instance, est préliée lors de la liaison de la référence à l'objet appelé.

Il ne reste plus qu'à tester la liaison de la référence de la classe appelée au code de la méthode appelée. Si elle n'est pas liée, une autre primitive du système est alors appelée.

Interface

L'interface de la machine à objets est composée des fonctions suivantes :

- Les fonctions utilisées lors de la compilation : **segobj_ClassCreate**, **segobj_CodeCreate**, **segobj_ModelCreate**. Elles permettent la création des segments produits par la compilation : les segments de classe, les segments contenant du code et les segments servant de modèles pour les opérations de création d'instance.
- Les fonctions appelées par l'environnement d'exécution par le code compilé des méthodes :
    - **segobj_ObjCreate**

      Création d'une instance d'une classe donnée en paramètre. La création est réalisée par copie du modèle d'instance de la classe.
    - **segobj_ObjDelete**

      Destruction d'une instance. La liaison de cette instance devient impossible, mais il peut subsister des vecteurs d'accès à cet objet dans des tâches.
    - **segobj_ObjCall**

      Cette fonction est appelée lorsqu'une référence à un objet doit être liée. Après la liaison de cette référence, elle peut déboucher sur un appel à la primitive de liaison de la référence à la méthode appelée, ou réaliser directement l'appel effectif de la méthode sur l'objet.
    - **segobj_MethCall**

      Cette primitive est appelée pour réaliser la liaison de la référence à la méthode appelée. Après cette liaison, elle réalise l'appel effectif de la méthode.
    - **segobj_ActObjCall**

      Lorsqu'un appel à un objet provient d'une autre tâche du même Domaine, la machine d'exécution doit réaliser le transfert de l'exécution d'une tâche (origine) à l'autre (destination) et doit relancer l'appel de l'objet appelé dans la tâche destination. Etant donné que le segment de liaison de l'objet appelant est dans la tâche origine, l'entrée de ce segment de liaison associée à la référence utilisée ne peut pas être utilisée dans la tâche destination. Cette fonction alloue donc un vecteur d'accès sur la pile et appelle la primitive *segobj_ObjCall ()* qui réalise l'appel de méthode, mais en utilisant un vecteur d'accès dans la tâche destination.
    - **segobj_ActTestObjCall**

      Si l'appel provient d'une tâche associée à un autre propriétaire d'objets, l'attribut de visibilité de l'objet appelé doit être à vrai pour que l'appel soit

autorisé. Dans ce cas, la machine d'exécution appelle cette fonction qui teste l'attribut de visibilité et réalise l'appel à l'objet.

- Les fonctions permettant de fixer la protection d'une application.

La figure Fig. 7.7 donne une vue globale de la gestion des objets. Les fonctions *segobj_ObjCall ()* et *segobj_MethCall ()* sont appelées pour lier les références aux objets et aux méthodes. Le cache de segments enregistre les liaisons de références résolues. Le cache de grappes enregistre les couplages de grappes déjà effectués.

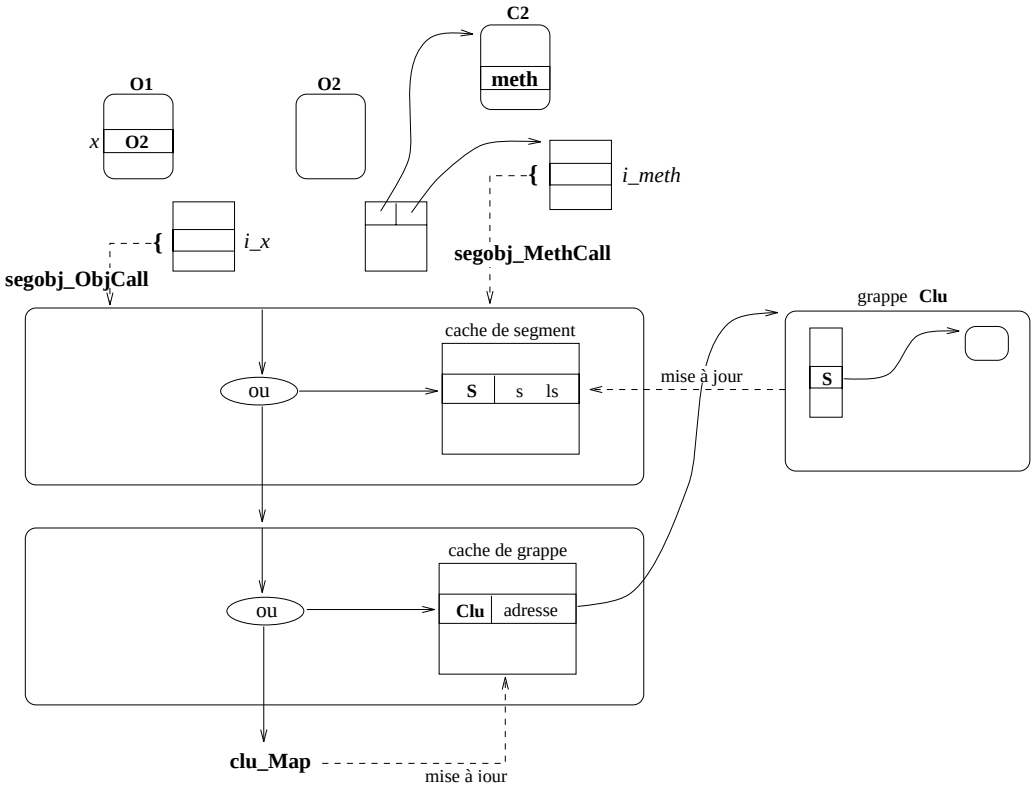

Fig. 7.7 : Vue globale de la machine à objets

Code généré

Le code généré (en langage C) pour l'appel de méthode à l'objet O2 dans la figure Fig. 7.7 est le suivant :

```
/* on prépare un bloc d'appel de méthode */
CB.ObjectHandle = &(Rblo[i_x]);
CB.Method = i_meth;
CB.Parameters = <paramètres>

/* on teste la liaison de l'objet appelé */
if (get_self(Rblo[i_x].s) != Rbo->x) then
    CB.ObjectRef = Rbo->x;
    segobj_ObjCall (&CB);
    return;
```

```
/* on teste la liaison de la méthode appelée */
if (Rblo[i_x].ls->ls[i_meth].s == NULL) then
    segobj_MethCall (&CB);
    return;

/* on appelle directement la méthode sans intervention
du système */
(Rblo[i_x].ls->ls[i_meth].s)  (&CB);
```

Le passage de paramètres aux méthodes appelées et aux primitives système a la même forme. Il est réalisé en passant l'adresse d'un bloc (appelé *CallBlock*) contenant les informations nécessaires. Cette interface, par rapport au passage de paramètre en extension, nous permet de transmettre de façon atomique un bloc d'appel de méthode lorsqu'il doit transiter entre des composants du système.

Un bloc d'appel contient :

- L'adresse du vecteur d'accès associé à la référence utilisée pour l'appel à l'objet (*ObjectHandle*). Il est mis à jour par la primitive *segobj_ObjCall ()* qui lie l'objet appelé. Il est aussi utilisé par la primitive *segobj_MethCall ()* pour adresser le segment de liaison de la classe, et dans la méthode appelée pour calculer les valeurs des registres *Rbo*, *Rblo* et *Rblc*.
- L'indice de la méthode (*Method*). Il est utilisé dans la primitive *segobj_MethCall ()* pour la liaison de la méthode appelée. Il est aussi utilisé dans la méthode appelée pour le calcul de la valeur du registre *Rblc*, donnant la base du segment de liaison du segment de code.
- Un pointeur (*Parameters*) sur la zone contenant les paramètres de la méthode appelée.
- La référence de l'objet appelé (*ObjectRef*), utilisé dans la primitive *segobj_ObjCall ()*.

L'appel de méthode est alors composé des étapes suivantes :

- Test de la liaison de l'objet appelé.

  La fonction *get_self ()* retourne la *SysRef* (dans l'entête) de l'objet désigné par le vecteur d'accès. Elle est comparée à la *SysRef* de l'objet appelé pour tester si la liaison est valide.

  En initialisant le vecteur d'accès d'une référence non liée avec un objet spécial contenant une *SysRef* nulle dans son entête, ce test détecte également le fait qu'une référence n'est pas liée.

  Si la référence à l'objet appelé n'est pas liée ou si cette liaison est obsolète, alors la primitive de liaison *segobj_ObjCall ()* est appelée.

- Test de la liaison de la méthode appelée.

  On teste la liaison de la référence à la méthode appelée dans le segment de liaison de la classe de l'objet appelé.

Il est possible d'améliorer l'appel de méthode en supprimant ce test. On initialise les vecteurs d'accès du segment de liaison de la classe avec l'adresse de la primitive *segobj_MethCall ()*. Le premier appel de la méthode appelle *segobj_MethCall ()*, qui remplace cette adresse par l'adresse de la méthode.

- Si ces deux références (à l'objet et à la méthode) sont liées, l'appel effectif peut être réalisé.

Le code généré pour la méthode appelée est le suivant :

```
method Meth (CB)
{
Rbo = CB->ObjectHandle->s;
Rblo = CB->ObjectHandle->ls;
Rblc = CB->ObjectHandle->ls->ls[i_meth].ls;

< code compilé >
}
```

Les registres Rbo, Rblo et Rblc sont initialisés en début de méthode.

VII.5 La machine d'exécution

Le but de la machine d'exécution est de fournir les primitives permettant la gestion des Domaines et des Activités.

Comme on l'a vu au chapitre V, un Domaine Guide est composé de plusieurs tâches de Mach pouvant s'exécuter sur des sites différents. Des objets utilisés par un Domaine peuvent être couplés dans des tâches différentes, afin de coupler ces objets sur des machines différentes, ou pour assurer l'isolation entre les propriétaires d'objets.

Une Activité Guide qui s'exécute dans une tâche est représentée par un processus léger de Mach dans cette tâche.

La machine d'exécution doit fournir les mécanismes permettant :

- La création et la destruction des Domaines et des Activités.

  Ces créations sont toujours réalisées de façon locale, c'est à dire sur le même site et dans la même tâche.

- Le changement de tâche d'exécution.

  Lorsque l'objet appelé se trouve dans une grappe qui ne peut pas être couplée dans la tâche courante, il est nécessaire de changer de tâche. Ce changement de tâche peut avoir lieu pour deux raisons :

  - pour aller dans une tâche associée au propriétaire de la grappe,
  - pour aller dans une tâche sur un site autorisé à coupler la grappe.

  La machine à grappes, lorsqu'elle refuse un couplage, retourne soit le propriétaire de la grappe (il faut alors aller dans une tâche associée à ce propriétaire), soit un site où le couplage sera autorisé (il faut alors aller dans une tâche sur ce site). Ces paramètres (le propriétaire et le site) sont alors donnés, avec les caractéristiques de l'appel de méthode à réaliser (*CallBlock*), à une primitive de la machine

d'exécution qui réalise l'appel de méthode à distance (entre deux tâches). Cet appel à distance transfère l'exécution dans la tâche destination et réalise l'appel d'objet dans cette tâche en appelant la primitive *segobj_ActObjCall ()*.

Si la tâche dans laquelle il faut aller s'exécuter n'existe pas, il faut la créer.

La machine d'exécution doit donc être en mesure de créer des tâches pour étendre les Domaines. Pour cette opération d'extension de Domaine, deux contraintes doivent être prises en compte :

- Il doit être possible de créer une tâche sur un site différent du site courant. Cette possibilité n'est pas fournie par le micro−noyau Mach. Il est donc nécessaire de gérer sur chaque site un démon de la machine d'exécution, qui reçoit les requêtes de création de tâches des autres sites et les réalise localement.
- A chaque tâche est associé un propriétaire d'objet et cette tâche ne peut alors coupler que des grappes appartenant à ce propriétaire. L'association entre une tâche et un propriétaire est un point très sensible dans notre schéma de protection. Cette association ne doit pas être falsifiable ; elle doit donc être gérée dans une tâche privilégiée. Elle est gérée dans les démons de la machine d'exécution. Ces démons coopèrent avec les gérants de mémoire pour contrôler le couplage des grappes.

Pour permettre à une Activité de changer de tâche, un processus lui est associé dans chaque tâche où elle s'est exécutée. Un port est associé à ce processus dans sa tâche. Un changement de tâche est réalisé par un envoi de requête d'exécution par message sur ce port. Le processus émetteur est alors suspendu en attente de la réponse et le processus récepteur exécute l'appel de méthode. Ce processus dans une tâche est appelé *extension* de l'Activité. Si un processus connaît le port de son extension dans une tâche, il peut changer de tâche par un simple envoi de message. S'il ne connaît pas ce port, il doit interroger le démon du site qui lui retournera ce port, en créant l'extension et le port si nécessaire.

Le dernier problème à résoudre concerne le test de l'attribut de visibilité. Lors d'un appel entre deux tâches T1 et T2, il faut authentifier de façon sûre le propriétaire associé à la tâche appelante et appeler la primitive adéquate (*segobj_ActObjCall ()* ou *segobj_ActTestObj-Call ()*) en fonction des propriétaires de T1 et T2, afin de contrôler l'attribut de visibilité de l'objet appelé si nécessaire. Etant donné que l'on ne fait pas confiance à la tâche appelante, on ne peut pas recevoir de celle−ci l'identité du propriétaire de la tâche appelante. De plus, on ne peut pas demander à T1 de s'authentifier auprès de T2 en envoyant son port d'identification, car cela permettrait à T2 de se faire ensuite passer pour T1.

Notre solution consiste à associer deux ports à chaque processus. Un port est destiné à recevoir les appels provenant du même propriétaire d'objets (port privé) et l'autre aux appels provenant des autres propriétaires (port public). Les appels sur le port privé provoquent l'appel à *segobj_ActObjCall ()* et les appels sur le port public provoquent l'appel à *segobj-_ActTestObjCall ()* (qui teste l'attribut de visibilité). Initialement, une Activité de T1 ne connaît aucun port lui permettant d'aller dans T2. Elle envoie alors une requête au démon de son site pour en obtenir un. Elle s'authentifie auprès du démon en envoyant le port d'identification de T1 et le démon lui retourne le port (public ou privé en fonction des propriétaires

des tâches T1 et T2) qui sera utilisé pour les appels suivants. Ainsi, une Activité dans la tâche T1 ne connaîtra qu'un seul port (privé ou public) lui permettant d'aller dans la tâche T2.

Interface

L'interface de la machine d'exécution est composée des fonctions suivantes :

- Les fonctions de gestion des Domaines : **job_Create** et **job_Destroy**, qui permettent la création et destruction de Domaines.
- Les fonctions de gestion des Activités : **act_Create**, **act_Exit** et **act_Join** qui permettent la création, la destruction et le rendez−vous d'Activités.
- La fonction d'appel entre tâches : **act_remCall**. Cette fonction est appelée pour transférer l'exécution d'une Activité dans une autre tâche. Les paramètres désignent le site et le propriétaire associés à la tâche destination, ainsi que les caractéristiques de l'appel de méthode (*CallBlock*) à réaliser dans cette tâche. Dans cette tâche, la primitive de la machine à objets *segobj_ActObjCall ()* est appelée.

VII.6 Interaction entre les machines

Nous proposons une vue globale des interactions entre les différentes machines sur la figure Fig. 7.8.

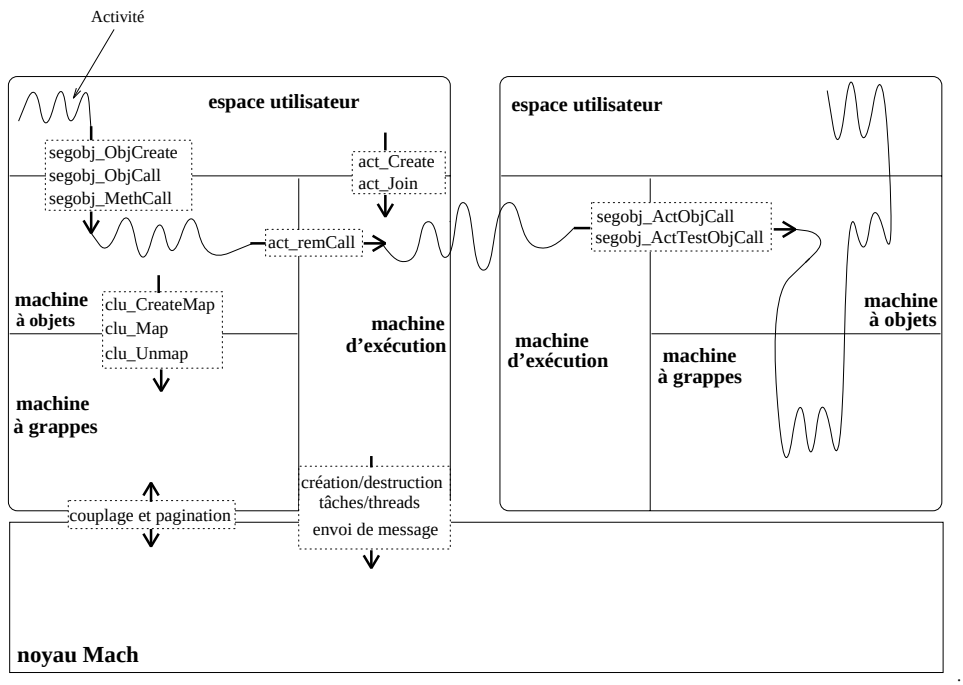

Fig. 7.8 : Interaction entre les différentes machines

Sur cette figure sont représentées deux tâches qui s'exécutent pour le compte du même Domaine. Le noyau du système Guide−2 est présent dans l'espace virtuel de chaque tâche. On y retrouve la machine à objets, la machine à grappes et la machine d'exécution.

Une activité, qui s'exécute en espace utilisateur dans la première tâche, entre dans la machine à objets pour traiter un défaut d'objet (*segobj_ObjCall*). Comme la grappe contenant l'objet appelé ne peut pas être couplée dans cette tâche, on appelle la machine d'exécution (*act_remCall*) pour aller dans une tâche autorisée à coupler la grappe. On se retrouve alors dans l'autre tâche dans la machine d'exécution, qui peut alors rédémarrer l'appel localement (*segobj_ActObjCall*). Une fois la grappe couplée dans le seconde tâche, l'objet est lié et l'exécution reprend dans l'espace utilisateur.

VII.7 Conclusion

Ce chapitre a présenté la réalisation du noyau Eliott. Ce noyau qui a été construit sur le micro−noyau Mach 3.0 est composé de trois sous−machines :

- La machine à grappes qui assure à la fois la persistance des grappes en mémoire de stockage et le couplage des grappes dans les tâches gérées en mémoire d'exécution. Elle joue également un rôle important pour la protection dans le système, en assurant qu'une grappe ne peut être couplée que dans une tâche associée à son propriétaire.
- La machine à objets qui fournit la notion de segments et l'adressage de ces segments, permettant la réalisation du modèle à objets primitif d'Eliott. L'adressage segmenté permet de ne pas interpréter tous les appels de méthode, dans le but d'en améliorer l'efficacité. Le contrôle des droits d'accès aux objets est réalisé lors de la liaison au premier appel et ne modifie pas le schéma d'adressage des objets.
- La machine d'exécution qui gère les structures d'exécution et qui permet l'appel de méthode à distance entre les tâches composant ces structures d'exécution. Elle permet également l'authentification de la tâche appelante permettant le test de l'attribut de visibilité.

Dans le chapitre suivant, nous présentons l'état d'avancement du système que nous avons réalisé, nous donnons une évaluation critique, fondée sur des mesures, des choix de conception du système et nous explorons les perspectives d'utilisation de la plate−forme réalisée.

Chapitre VIII

Evaluation

Pour évaluer notre système, nous allons tout d'abord présenter l'état courant des travaux dans Guide−2 (section VIII.1), puis proposer une évaluation de l'adéquation du micro−noyau Mach 3.0 pour sa conception (section VIII.2).

Le noyau Eliott a été, au cours de sa réalisation, équipé de capteurs permettant de mesurer l'efficacité des mécanismes fournis et d'évaluer par des statistiques le bien fondé des hypothèses faites lors de la conception sur le comportement des applications. Les expériences d'utilisation (décrites en section VIII.3) nous ont donc permis d'évaluer le système en collectant ces mesures pour chaque application supportée. Nous présentons les résultats de cette évaluation (section VIII.4).

VIII.1 Etat courant

Nous présentons l'état du noyau Eliott en juillet 1993.

Il est opérationnel sur un réseau de PC 80386 et 80486. Ces machines sont démarrées avec le système OSF−1 construit sur le micro−noyau Mach 3.0.

Le noyau Eliott n'est donc pas complètement indépendant d'OSF−1 et certaines parties du noyau sont même réalisées en utilisant des fonctions d'OSF−1. On peut citer notamment le fait que les tâches de Mach que nous utilisons sont en réalité des processus du système OSF−1 (qui sont eux−mêmes réalisés par des tâches de Mach). Cette réalisation nous a permis notamment d'avoir accès à des services fournis par OSF−1 (comme les entrées−sorties), que nous aurions du réécrire si nous avions directement utilisé des tâches.

Le noyau Eliott est composé d'une part de code et de données présents dans l'espace virtuel de chaque tâche et d'autre part de démons qui sont des tâches privilégiées du système. Nous détaillons quelques aspects du système actuellement disponible.

La machine à grappes

La machine à grappes est composée d'une partie librairie présente dans chaque tâche et de gérants de mémoire qui sont des paginateurs de Mach. Lors du démarrage d'une machine, un gérant de mémoire est créé sur ce site et les volumes spécifiés dans une table de montage sont montés par ce gérant de mémoire.

Dans la version actuelle, les gérants de mémoire permettent le partage de grappes entre machines différentes, en assurant une cohérence stricte des données suivant un protocole du type n lecteurs ou (exclusif) un rédacteur.

Le service de stockage est très primitif : il est simplement réalisé en stockant les grappes dans des fichiers du système Unix (OSF−1). L'architecture de la machine à grappes prévoit la possibilité de réutiliser des serveurs de stockage existants. Il reste à en récupérer.

Des outils d'administration de la machine à grappes permettent de modifier des attributs des grappes et de fixer notamment si celles−ci peuvent être partagées sur plusieurs sites.

La machine à objets

La machine à objets est uniquement composée de librairies présentes dans chaque tâche. Elle permet la définition de classes et l'exécution d'applications compilées par le compilateur du langage Guide.

Le modèle à objets fourni par le noyau est utilisé par le compilateur pour mettre en œuvre le modèle à objets du langage Guide, mais les mécanismes de protection, bien que réalisés, ne sont pas encore utilisés par des outils qui devraient permettre au concepteur d'application de fixer les attributs relatifs au contrôle d'accès associés aux instances (listes d'accès) et aux classes (vues). Un objectif clé est de continuer le travail entrepris sur la protection, soit en enrichissant les langages supportés, soit en offrant des outils de configuration, permettant de programmer la protection des applications développées dans notre environnement.

De même, le mécanisme de migration d'objet n'est pas exploité par des outils de configuration d'applications pour lesquels il a été conçu.

La machine d'exécution

La machine d'exécution est composée d'une partie librairie présente dans chaque tâche et de démons (un par site) qui sont créés lors du démarrage de chaque site.

La machine d'exécution permet la gestion de Domaines composés de plusieurs tâches pouvant s'exécuter sur plusieurs sites. Les Activités dans un Domaine peuvent changer de tâche d'exécution par appel d'objets entre tâches, quels que soient les sites de ces tâches.

Des outils d'observation de la machine d'exécution ont été développés et permettent de suivre le déroulement d'une application répartie, en montrant les composants des Domaines (tâches) et des Activités (processus légers) créés pendant l'exécution.

Les compilateurs

Le compilateur Guide, qui a été réalisé dans le cadre de Guide−1, a été adapté à la génération de code sur le noyau Eliott. Ce compilateur permet de développer des applications en définissant des classes. La génération de code pour l'héritage de classe n'est pas encore disponible.

Le modèle à objets de base fourni par le noyau Eliott doit permettre de réaliser des compilateurs pour d'autres langages orientés−objets persistants. Des étudiants de DESS travaillent actuellement sur la réalisation d'un compilateur pour une extension du langage C++.

Le développement d'Eliott

Le noyau Eliott est le fruit du travail d'une équipe. Les spécifications ont duré environ dix huit mois. Le développement a commencé il y a deux ans et il se poursuit. Eliott est composé d'environ 40 000 lignes de code écrit en langage C. La gestion de ces sources est assurée en utilisant le logiciel de gestion de versions et de configurations Adèle [Verilog93]. Des services d'administration ont également été développés afin que le système soit utilisable dans de bonnes conditions.

Etat général

Le noyau Eliott et le compilateur Guide sont actuellement utilisés par des éudiants de DEA et de Thèse qui l'utilisent comme plate−forme pour mener des expériences dans le cadre de leurs activités de recherche. Des applications ont également été développées en langage Guide à des fins de démonstrations. L'environnement Guide a notamment participé à des démonstrations :

- ENTERPRISE−93 à Boston,
- Revue des Projets de Recherche Coordonnés (PRC) à Paris,
- Ecole d'été Bull−IMAG−INRIA à Autrans.

Le système n'a pas encore été utilisé de façon intensive, sur un grand nombre de stations de travail (plus de trois) et pendant une longue période (plusieurs jours).

VIII.2 Adéquation de Mach 3.0

Les deux prototypes développés respectivement sur Unix et sur Mach 3.0 ont montré d'une part que le système Unix se prêtait mal à la réalisation d'un système comme Guide et d'autre part que le micro−noyau Mach fournissait une plate−forme bien plus adaptée [Chevalier93a]. Les avantages principaux sont les suivants :

- Le partage de l'espace d'adressage d'une tâche par plusieurs flots d'exécution (processus légers) permet de réaliser plus simplement les structures d'exécution de Guide. En effet, le modèle d'exécution de Guide peut être vu comme une version répartie de modèle d'exécution de Mach. Un Domaine est un espace d'adressage potentiellement réparti dans lequel plusieurs Activités peuvent s'exécuter en concurrence. Un Domaine et ses Activités sont donc naturellement réalisés par un ensemble de tâches et de processus légers sur les sites où le Domaine est représenté.
- Un port de Mach est un nom associé à une boîte aux lettres. Ce nom peut être utilisé indépendamment de la localisation de l'émetteur et du récepteur, ce qui résout en partie les problèmes relatifs à la répartition des structures d'exécution. Dans la réalisation du noyau Eliott, il a été possible de développer et mettre au point le système sur une seule machine, car les primitives d'envoi et de réception de messages entre des tâches sur la même machine ou sur des machines différentes ont la même interface. Le passage d'une version centralisée à une version répartie a nécessité peu d'efforts.

- Une partie de la gestion de la mémoire virtuelle est réalisée en dehors du noyau par un processus utilisateur (paginateur), ce qui nous a permis de réaliser dans de meilleures conditions notre gestion du partage et de la persistance. Les paginateurs nous ont permis de gérer distinctement l'adressage au niveau de l'objet, le partage au niveau de la grappe et les entrées−sorties au niveau de la page.
- Mach permet de faire cohabiter plusieurs systèmes sur la même machine. OSF−1 et Eliott en sont des exemples.

Néanmoins, quelques aspects nous ont gênés dans notre conception.

- La gestion des structures d'exécution aurait pu être simplifiée si nous avions disposé de la possibilité de créer des tâches sur un site depuis un site distant.
- La notion de groupe de ports permettant la diffusion de message n'existe pas. Elle pourrait s'avérer utile pour gérer les structures d'exécution, ou pour réaliser des fonctions de tolérance aux pannes.

Enfin, la notion de port offre à la fois des avantages et des inconvénients. Les ports de Mach sont désignés par des noms locaux dans les tâches et ils ne peuvent être partagés que par envoi explicite dans un message. Cette caractéristique a permis d'assurer la sûreté de nos mécanismes de protection. Les ports sont utilisés comme des capacités, à la fois pour contrôler les changements de tâches dans un Domaine et pour contrôler les droits de couplage des grappes par ces tâches.

Par contre, le fait que les ports ne puissent être partagés implicitement dans de la mémoire virtuelle partagée nous a gênés pour la réalisation d'outils de synchronisation. En effet, pour réaliser des sémaphores, il faut être en mesure de partager un compteur et une file d'identificateurs de processus en attente sur le sémaphore. Les identificateurs des processus en attente sont utilisés pour réveiller un processus qui prend le sémaphore. Si on veut réaliser des sémaphores en mémoire partagée entre les tâches en utilisant les primitives de suspension et de réveil des processus légers de Mach, il faut pouvoir stocker en mémoire partagée des identificateurs de processus légers (donc des ports), afin d'être capable de les réveiller ; or, Mach ne permet pas le partage implicite de ports. La seule solution pour réaliser un sémaphore avec les ports de Mach consiste à gérer ce sémaphore dans une tâche (un serveur), chaque primitive sur le sémaphore se traduisant par une requête (RPC) à ce serveur qui contient les identificateurs de toutes les activités en attente sur le sémaphore. Plutôt que d'utiliser cette solution coûteuse, nous avons opté pour l'utilisation des *spin_lock* de Mach qui réalisent une attente active sur un verrou, car un *spin_lock* peut être placé en mémoire partagée. Une entrée en section critique se traduit par une opération *Test_and_Set* sans envoi de message à un serveur.

VIII.3 Expériences d'utilisation

Différentes applications ont été développées en langage Guide. L'objectif du développement de ces applications est double :

- Les démonstrations.

  Le seul moyen de montrer que notre système marche est de faire s'exécuter des applications.

- Les mesures.

  Evaluer un système ainsi que les choix faits lors de sa conception ne se limite pas à mesurer l'efficacité de l'appel de méthode, même si ce paramètre est loin d'être négligeable. Il nous paraît primordial d'être capable d'observer les réactions du système lorsque des applications (différentes de la boucle de 1000000 appels de méthode sur le même objet) s'exécutent.

Nous décrivons ici trois applications (dont une est de taille importante) qui ont été utilisées pour les mesures données dans la section suivante.

Les tours de Hanoi

C'est le premier programme écrit en langage Guide. Il y a trois objets qui représentent des piquets et dix objets qui représentent des disques de diamètres décroissants empilés sur le premier piquet. Le but est de faire passer tous ces disques sur le troisième piquet avec deux contraintes :

- les disques sont déplacés un par un,
- on ne peut empiler un disque sur un disque de diamètre plus petit.

L'algorithme est récursif.

Pour tester le fonctionnement réparti du système, nous avons réalisé ce programme en imposant l'utilisation de chaque objet piquet sur un site différent. Ainsi, un grand nombre d'appels à distance sont exécutés. Ce programme est utilisé dans les démonstrations pour illustrer nos structures d'exécution.

Le benchmark de Cattell

Ce benchmark [Cattell92] est utilisé dans le domaine des bases de données orientées−objets. Il permet d'en mesurer l'efficacité et notamment la taille de la base générée, le coût d'une recherche à l'aide d'un index, mais également le coût de l'appel de méthode. Nous avons donc programmé ce benchmark pour mesurer l'appel de méthode, mais aussi pour observer les réactions du système.

Il se compose d'un grand ensemble d'objets (environ 10000). Chaque objet contient des données, ainsi que trois références vers trois objets choisis aléatoirement. On définit ainsi un graphe d'objet. Le but est de mesurer le coût d'un parcours du graphe à partir d'un objet jusqu'à sept niveaux de profondeur, en appelant une méthode vide sur chaque objet visité (3280 appels).

L'intérêt de ce benchmark est de mesurer l'appel de méthode dans des conditions plus proches du fonctionnement normal. Les objets ont un état et des variables sont passées en paramètre. De plus, ce programme permet de mettre en évidence le fait

que les performances du système dépendent directement du taux de réutilisation des références aux objets. Cet aspect est détaillé en section VIII.4.

Un tableur réparti

Le but de cette application est de fournir un tableur coopératif réparti. Une table est composée de cellules. Une cellule est une case mémoire pouvant contenir un entier ou une formule mathématique référençant d'autres cellules dans la même table ou dans une table différente. Les tables (donc les cellules) sont toutes persistantes.

Un exemple d'utilisation de ce tableur consiste à mettre en œuvre la gestion des stocks d'une chaîne de magasins. Chaque magasin gére ses propres stocks dans une table locale et la gestion globale de la chaîne de magasins est assurée par une table globale qui fait référence aux cellules des tables locales.

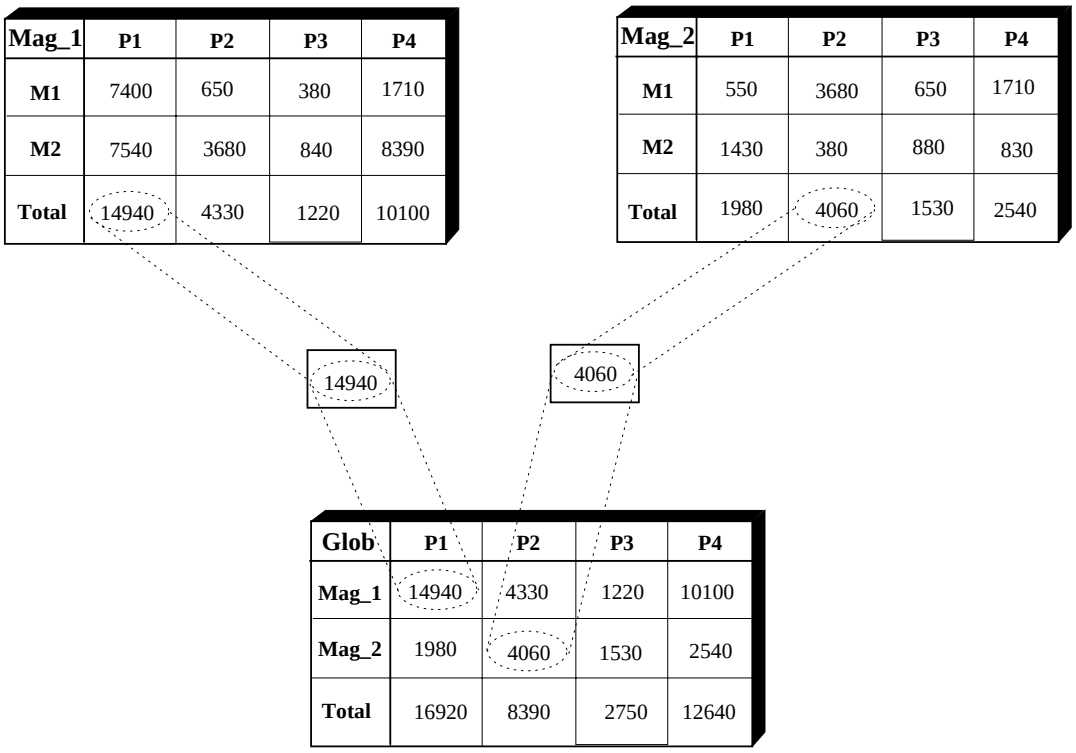

Fig. 8.1 : Le tableur coopératif réparti

Cette application est illustrée sur la figure Fig. 8.1. Deux magasins *Mag_1* et *Mag_2* gèrent chacun une table dans laquelle ils enregistrent leurs stocks respectivement pour les produits P1 à P4 et pour les marques M1 et M2. Le centre de gestion des magasins (*Glob*) gère une table donnant pour chaque produit les stocks des deux magasins ainsi que les stocks

cumulés. Les cellules des magasins qui indiquent les totaux par produits sont partagées avec la table globale.

Dans cette application, le partage intervient à deux niveaux :

- Aux niveaux des cellules, puisque les cellules peuvent être partagées entre les tables.
- Aux niveaux des tables, puisqu'une personne travaillant au centre de gestion des magasins peut consulter à la fois la table globale et les tables locales des deux magasins.

VIII.4 Mesures

Nous présentons les mesures qui ont été faites sur le noyau Eliott [Chevalier93b]. Ces mesures concernent principalement la machine à objets. Deux types de mesures ont été effectués :

- Les mesures d'efficacité.

  Ces mesures consistent en des appels répétés à des services du système pour en mesurer le coût. Nous avons ainsi mesuré le coût d'un appel de méthode dans les différents cas pouvant se produire.

- Les statistiques.

  Notre mécanisme d'adressage des objets repose sur la supposition que l'ensemble des variables qui contiennent des références d'objets, utilisées pour les appels aux objets, est petit. Nous supposons que lorsqu'un chemin d'accès à un objet est établi, il est souvent réutilisé. Nous avons donc inclus dans le système des capteurs qui calculent le nombre d'utilisations de chaque variable et qui nous fournissent le pourcentage des appels qui ne sont pas interprétés et qui profitent donc de notre mécanisme de liaison au premier appel.

Mesures d'efficacité

La table suivante donne des mesures effectuées sur une machine Bull−Zenith P.C. 486 (33MHz).

Opération mesurée		Coût
Appel direct à un objet	(1)	4.4 µs
Défaut d'objet (cache de segment à jour)	(2)	22 µs
Défaut d'objet (cache de grappe à jour)	(3)	55 µs
Défaut d'objet (avec couplage de grappe)	(4)	10 ms
Défaut de méthode (cache de segment à jour)	(5)	35 µs
Appel entre deux tâches (même machine)	(6)	980 µs
Appel entre deux tâches (machines différentes)	(7)	4 ms

Un appel direct à un objet (1) se produit lorsque le vecteur d'accès associé à la référence à l'objet appelé est à jour. Dans ce cas, l'appel n'est pas interprété par le noyau.

Le coût d'un défaut d'objet lorsque le segment réalisant cet objet est présent dans le cache de segment (2) ne nécessite que la consultation du cache et la mise à jour du vecteur d'accès en fonction de la protection associée à l'objet.

Lorsque le segment réalisant l'objet appelé n'est pas dans le cache de segments, le cache de grappes est consulté. Si le cache de grappes est à jour (3), celui−ci retourne l'adresse de couplage de la grappe, le segment est recherché dans cette grappe et le cache de segments et mis à jour. Le vecteur d'accès peut alors être affecté comme précédemment.

Lorsque le cache de grappes n'est pas à jour (4), la grappe est couplée dans la tâche courante et le cache de grappes est mis à jour. Le coût de la liaison de la référence n'est pas visible, puisque le coût de l'opération de couplage est bien plus grand.

On donne en (5) le coût de la liaison d'une référence à une méthode, lorsque le segment contenant le code de la méthode est dans le cache de segments. La définition des vues de la classe est consultée et le vecteur d'accès dans le segment de liaison de la classe est mis à jour si la vue concernée l'autorise. C'est la consultation de la vue dans la classe qui explique la différence entre les mesures (2) et (5). Cette différence peut être considérée comme le coût dû à la vérification du droit d'appel à la méthode, qui n'a lieu qu'au premier appel à une méthode.

Dans le cas le plus défavorable, un défaut d'objet est suivi d'un défaut de méthode, ces deux défauts nécessitant le couplage de deux grappes.

Pour les appels entre objets dans des tâches différentes, les coûts sont très liés aux coûts des échanges de message entre tâches sur la même machine et entre différentes machines. Les mesures initialement effectuées entre différentes machines avec Mach 3.0 n'étaient pas très bonnes ; ces mauvaises mesures s'expliquent par le fait que les protocoles de communication sont en dehors du noyau Mach, ce qui implique de nombreux changements de tâches à chaque envoi de message. Nous avons ensuite utilisé une version dérivée de Mach 3.0 appelée Mach 3.0/NORMA, qui permet de considérer un réseau de stations de travail comme étant un multiprocesseur. Les communications sont intégrées dans le noyau NORMA. Cette version nous a permis d'obtenir une bien meilleure efficacité (6−7).

Pour évaluer ces performances, on peut effectuer différentes comparaisons. Nous prenons ici l'appel de procédure en C comme étalon et nous comparons le coût d'un appel de méthode direct avec Eliott (non interprété), le coût d'un appel de méthode en C++ sur un système Unix et le coût d'un appel de méthode local dans le système Emerald [Jul88].

Appel de procédure C	1
Appel de méthode C++	1.7
Appel de méthode Eliott	4.9
Appel de méthode Emerald	1.5

La différence entre les temps d'Emerald et de C++ et ceux d'Eliott se justifie par le fait que dans Emerald et C++, tous les objets sont désignés par une adresse dans un espace virtuel unique. Dans le cas de C++, les objets ne sont pas partagés entre différents processus. Dans le cas d'Emerald, tous les processus d'Emerald sont réalisés par des processus légers s'exécutant dans un processus Unix unique sur le site courant et l'isolation des objets est assurée par le langage Emerald.

Eliott permet le partage d'objets entre des processus s'exécutant dans des domaines de protection différents ; il fournit également des mécanismes permettant de contrôler l'accès aux objets. Les solutions fondées sur la gestion d'un espace virtuel partagé unique, déjà étudiées dans plusieurs projets de recherche [Chase92b][Inohara93], ouvrent de nouvelles perspectives, car elles permettront de cumuler un appel de méthode très rapide ainsi que des mécanismes de protection.

Comportement des applications

L'objectif de ces statistiques est de confirmer ou d'infirmer la supposition de localité d'utilisation des variables désignant les objets appelés. Sur les applications décrites auparavant, nous avons obtenu les résultats présentés dans la table suivante.

Application	**Pourcentage d'appels avec une référence d'objet liée**
Tableur	85–97 %
Tours de Hanoi	37 %
Benchmark de Cattell (1000 objets)	91 %
Benchmark de Cattell (5000 objets)	71 %
Benchmark de Cattell (8000 objets)	58 %

Avant de commenter ces chiffres, il faut noter que nous avons réalisé les mêmes mesures pour les références aux méthodes, lorsqu'elles sont liées de façon paresseuse (sans préliaison). Nous avons observé que 99% des appels de méthode ne provoquent pas de défaut de méthode, pour les applications décrites dans cette évaluation. La liaison d'une référence à une méthode n'est en effet réalisée qu'une seule fois pour toutes les instances de la classe. Ce résultat est très encourageant, bien qu'il doive être tempéré par le fait que le nombre de classes définies dans ces

applications est petit. Des mesures sur des applications plus importantes feront l'objet de travaux futurs.

Pour les références aux objets, la localité des références utilisées semble également très bonne pour l'application Tableur, bien qu'elle dépende du type de la session de travail (création de tables, consultations, modifications).

Pour le benchmark de Cattell, la localité dépend naturellement du nombre d'objets de la base. Lorsque l'on augmente le nombre d'objets de la base, la probabilité que deux objets référencent le même objet dans le graphe diminue.

Cependant, la localité des références utilisées n'est pas forcément très forte. Lorsqu'une application déclare une variable temporaire sur la pile, un vecteur d'accès est associé à cette référence sur la pile et il est lié lors de la première utilisation. Pour l'application des Tours de Hanoi, un nombre important de références sont passées en paramètres (donc sur la pile) à des appels récursifs de méthode. Un vecteur d'accès sur la pile est alors utilisé et cette liaison est perdue lorsque les paramètres sont dépilés. Ainsi, dans l'exemple des Tours de Hanoi, de nombreuses références ne sont utilisées qu'une seule fois.

Notons que, lorsqu'une *SysRef* est passée en paramètre d'un appel de méthode, nous avons choisi de ne pas passer le vecteur d'accès en paramètre, car l'appel de méthode peut provoquer un changement de tâche dans laquelle le vecteur d'accès ne serait plus valide. Une solution envisageable pourrait consister à passer le vecteur d'accès en paramètre (pour profiter de la liaison) et à l'invalider lors d'un changement de tâche.

Globalement, ces premières mesures confirment les hypothèses faites dans la phase de conception du système, à savoir que :

- Un schéma de liaison au premier appel permet de réduire de façon significative le coût de l'appel de méthode pour les appels suivants.
- Lorsqu'un chemin d'accès à un objet est mis en place par le mécanisme de liaison d'une référence, il est souvent réutilisé.

Chapitre IX

Conclusion

L'objectif du projet Guide est la conception d'un système d'exploitation réparti, permettant le développement d'applications coopératives, structurées en termes d'objets partagés persistants. Après une première version (Guide−1) développée sur le système Unix de 1987 à 1990, une seconde a été réalisée sur le micro−noyau Mach 3.0. Mon travail de thèse s'est situé dans la conception et la réalisation de ce nouveau prototype (Guide−2) et concerne plus particulièrement les mécanismes d'adressage et de protection des objets.

Le principal résultat du projet Guide est la réalisation d'un noyau de système réparti opérationnel sur un ensemble de stations de travail. Ce noyau appelé Eliott fournit le support nécessaire à l'exécution d'applications réparties programmées à l'aide du langage orienté−objet Guide. Le noyau Eliott est constitué de trois composants indépendants les uns des autres : la machine à grappes dont le but est de gérer le partage des objets regroupés dans des grappes, la machine d'exécution qui gère des structures d'exécution réparties et la machine à objets qui permet un adressage efficace des objets par les structures d'exécution. Le système est ouvert dans le sens où il permet au niveau de la machine à grappes la récupération de serveurs de stockage déjà existants et il fournit au niveau de la machine à objets un modèle de base pour le support de différents langages orientés−objets. Des mécanismes de protection (isolation) sont intégrés à cette machine virtuelle, ce qui autorise le support de langages sans faire d'hypothèses sur la sûreté du code généré par les compilateurs. Cette ouverture peut être opposée à la solution de Guide−1 où le système était construit pour le support d'un unique langage (supposé sûr) et où un unique service de stockage était utilisé.

Mon travail de thèse s'est focalisé sur le partage, l'adressage et la protection des objets.

Le principal objectif était de permettre le partage des objets gérés par le système entre des structures d'exécution pouvant s'exécuter sur des machines différentes. Pour ce faire, deux mécanismes ont été réalisés. Le premier permet le partage par copies multiples, qui est réalisé à l'aide des paginateurs de Mach. Il est ainsi possible de coupler des objets sur des sites différents et le système assure la cohérence des données partagées. L'autre technique consiste à changer de site d'exécution pour aller partager l'objet désiré sur un même site. Ces deux mécanismes sont complémentaires. En effet, pour les objets souvent modifiés, il est préférable de les partager sur un seul site pour ne pas payer le coût du maintien de la cohérence. Par contre, les objets rarement modifiés peuvent être partagés par copies multiples. Pour réduire le nombre d'opérations de couplage et d'entrée−sortie, les objets sont

regroupés dans des grappes et la grappe est l'unité de couplage par les structures d'exécution.

Le deuxième résultat important de ce travail concerne l'adressage des objets. Il fallait permettre un adressage efficace des objets par des structures d'exécution, sachant que ces objets sont potentiellement partagés et persistants. Nous avons réalisé un mécanisme d'adressage où seuls les premiers appels sont interprétés. Nous sommes parvenus à implanter des objets partagés et persistants avec un coût raisonnable par rapport à une exécution mono−utilisateur sans persistance : le coût d'un appel de méthode n'est que trois fois moins rapide que celui d'un appel en C++ sur le système Unix où le partage et la persistance sont laissés à la charge de l'utilisateur ; le coût de l'accès aux variables d'état de nos objets est du même ordre.

Enfin, une attention particulière a été portée aux problèmes relatifs à la protection. Le but était d'une part d'assurer une isolation des objets et des structures d'exécution afin de limiter la portée d'une erreur dans une méthode et d'autre part de permettre l'expression de droits d'accès associés aux objets pour programmer la protection des applications. L'isolation est assurée par le fait que les Domaines de Guide ne partagent pas de tâches (domaines de protection) et par le fait que dans un Domaine, une tâche est associée à chaque propriétaire d'objet, ce qui semble être un bon compromis entre une isolation totale (un domaine de protection par objet) et une isolation nulle (tous les objets dans le même domaine de protection). Le contrôle d'accès a été intégré au mécanisme d'adressage sans en modifier le coût : le contrôle est réalisé au premier appel.

Les perspectives à court terme de mon travail de thèse consistent en des extensions au système Guide−2 et en des expérimentations sur le prototype réalisé.

En ce qui concerne la machine à grappes, deux extensions sont déjà en cours de réalisation. La première consiste à modifier le protocole de cohérence utilisé lors d'un partage des grappes par copies multiples. Le protocole actuellement mis en œuvre assure une cohérence stricte des données partagées. Un protocole de cohérence faible, permettant la simultanéité d'une écriture et de plusieurs lectures, est en cours de développement. La deuxième extension concerne la récupération de serveurs de stockage. L'architecture de la machine à grappes a prévu la possibilité de réutiliser des serveurs de stockage existants, apportant des services autres que le stockage simple, comme par exemple une disponibilité accrue. Nous nous proposons de réaliser un service de stockage dupliqué ; ce travail fait actuellement l'objet d'une thèse[Chevalier]. Nous étudions également, dans le cadre du projet BROADCAST auquel nous participons, la possibilité de récupérer un service de stockage externe réalisé dans le projet Pegasus [Stabell93] à l'Université de Twente.

Un autre aspect qui doit être étudié est la coopération entre les applications et le système, dans le but d'exploiter pleinement les mécanismes proposés. Le regroupement des objets dans les grappes, s'il doit permettre une meilleure efficacité, doit être réalisé intelligemment. Il faut être capable de regrouper les objets en fonction du comportement des applications. Ces données sur le comportement des applications peuvent provenir soit du système soit des compilateurs utilisés. Il faut envisager une coopération entre les compilateurs et le système afin de modéliser le comportement des applications. Une modélisation du comportement des

applications peut également être utilisée pour choisir le mécanisme utilisé pour partager les grappes. Une grappe peut en effet être partagée par copie multiple entre des machines différentes, ou en localisant ce partage sur un site unique en utilisant l'appel de méthode à distance. Il doit être possible de choisir judicieusement le mode de partage en fonction du type des objets (objets modifiés fréquemment ou rarement), le problème étant de choisir la meilleure stratégie pour chaque grappe.

Enfin, des mécanismes sont fournis par le système pour permettre la protection des applications par contrôle d'accès aux objets, mais aucun outil n'existe à l'heure actuelle, permettant de définir la protection des applications. Un objectif clé est de continuer le travail entrepris sur la protection, soit en enrichissant les langages supportés, soit en offrant des outils de configuration permettant de programmer la protection des applications développées dans notre environnement. Il faudra ensuite mener une expérimentation plus large à travers le développement d'applications protégées en grandeur réelle.

Pour résumer, mis à part quelques extensions en cours de réalisation, notre objectif à court terme est double :

- permettre la coopération entre les compilateurs et le système pour améliorer l'utilisation des mécanismes proposés,
- supporter des applications plus importantes, permettant d'évaluer notre système dans des conditions réelles.

A plus long terme, la recherche en système d'exploitation est influencée par l'évolution technologique : apparition de nouveaux processeurs RISC rapides avec capacités étendues d'adressage virtuel (64 bits) et de réseaux de communication très rapides (débits au delà du Gbit/s).

Avec les processeurs 64 bits, l'ensemble des objets accessibles sur un réseau local peuvent être stockés dans un espace virtuel unique, ce qui simplifie la gestion des références entre les objets. Le problème fondamental à résoudre est alors celui de la protection. L'exploitation des grands espaces virtuels fait déjà l'objet de projets de recherche [Chase92a][Heiser93][Inohara93] et d'études préliminaires dans le projet Guide.

La disponibilité de réseaux très rapides permet d'envisager des solutions nouvelles pour la gestion de la pagination et pour le partage d'information. Le partage par copies multiples devrait se généraliser, tout comme la pagination à distance.

La tolérance aux défaillances a été largement laissée de coté dans Guide. Cependant, l'une des motivations premières de la répartition est justement de permettre un fonctionnement dégradé dans le cas de la déconnection volontaire ou involontaire de machines. Des extensions aux micro−noyaux devraient être introduites pour permettre une gestion efficace des mécanismes de reprise (transactions).

Nous souhaiterions enfin montrer la validité de nos idées en menant une expérimentation à grande échelle, avec la mise en place d'une plate−forme permettant d'étudier des systèmes répartis en vraie grandeur (plusieurs dizaines de stations de travail), sur des applications avancées.

Bibliographie

[Accetta86]

M.J. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian et M. Young, ''Mach: A new kernel foundation for Unix development'', *Proceedings of the USENIX 1986 Summer Conference*, pp. 93–112, Eté 1986.

[Bal89]

H.E. Bal, M.F. Kaashoek et A.S. Tanenbaum, ''A distributed implementation of the shared data–object model'', *USENIX Workshop*, Octobre 1989.

[Bal92]

H.E. Bal, M.F. Kaashoek et A.S. Tanenbaum, ''Orca: A Language for Parallel Programming of Distributed Systems'', *IEEE Transactions on Software Engineering*, 18(3), pp. 190–205, Mars 1992.

[Balter91]

R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville et G. Vandôme, ''Architecture and implementation of Guide, an object–oriented distributed system'', *Computing Systems*, 4(1), pp. 31–67, Hiver 1991.

[Banâtre91]

J.P. Banâtre et M. Banâtre, *Les systèmes distribués, Concepts et expérience de Gothic*, InterEditions, 1991.

[Bernabeu88]

J.M. Bernabeu–Auban, P.W. Hutto, M.Y.A. Khalidi, M. Ahamad, W.F. Appelbe, P. Dasgupta, R.J. LeBlanc et U. Ramachandran, *The Architecture of Ra: A Kernel for Clouds*, (GIT–ICS–88/25), School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, 1988.

[Black85]

A.P. Black, ''Supporting distributed applications: experience with Eden'', *10th ACM Symposium on Operating System Principles, SIGOPS Operating Systems Review*, 19(5), pp. 181–193, Decembre 1985.

[Black86]

A.P. Black, N. Hutchinson, E. Jul et H. Levy, ''Object structure in the Emerald

system'', *Proceedings of the 1st ACM Conference on Object−Oriented Systems, Languages and Applications (OOPSLA)*, Septembre 1986.

[Boyer91a]

F. Boyer, J. Cayuela, P.Y. Chevalier, A. Freyssinet et D. Hagimont, ''Supporting an object−oriented distributed system: experience with Unix, Chorus and Mach'', *Proceedings of the 2nd Symposium on Experience with Distributed and Multiprocessor Systems (SEDMS)*, Mars 1991.

[Boyer91b]

F. Boyer, ''A Causal Distributed Shared Memory Based on External Pagers'', *Proceedings of the 2nd Usenix Mach Symposium*, pp. 41−59, Novembre 1991.

[Cattell92]

R. G. G. Cattell et J. Skeen, ''Object Operations Benchmark'', *ACM Transactions on Database Systems*, 17(1), pp. 1−31, Mars 1992.

[Chase89]

Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy et Richard J. Littlefield, *The Amber System: Parallel Programming on a Network of Multiprocessors*, (TR 89−04−01), Department of Computer Science and Engineering, University of Washington, Seattles, Avril 1989.

[Chase92a]

Jeffrey S. Chase, Henry M. Levy, Miche Baker−Harvey et Edward D. Lazowska, *Lightweight Shared Objects in a 64−Bit Operating System*, (TR 92−03−09), Department of Computer Science and Engineering, University of Washington, Seattles, Mars 1992.

[Chase92b]

Jeff Chase, Henry Levy, Miche Baker−Harvey et Ed Lazowska, ''Opal: A Single Address Space System for 64−bit Architectures'', *Proceedings of the 3rd Workshop on Workstation Operating Systems (WWOS)*, pp. 80−85, Avril 1992.

[Chase92c]

Jeffrey S. Chase, Henry M. Levy, Miche Baker−Harvey et Edward D. Lazowska, *How to Use a 64−Bit Virtual Address Space*, (TR 92−03−02), Department of Computer Science and Engineering, University of Washington, Seattles, Mars 1992.

[Chevalier]

P.Y. Chevalier, *Persistence et disponibilité dans les systèmes répartis*, Thèse de doctorat à paraitre, Université Joseph Fourier, Grenoble I.

[Chevalier92a]

P.Y. Chevalier, ''A Replicated Object Server for a Distributed Object−Oriented System'', *11th Symposium on Reliable Distributed Systems*, Octobre 1992.

[Chevalier92b]

P.Y. Chevalier, D. Hagimont, S. Krakowiak et X. Rousset de Pina, ‘‘System Support for Shared Objects’’, *Proceedings of the 5th ACM European SIGOPS Workshop*, Septembre 1992.

[Chevalier93a]

P.Y. Chevalier, A. Freyssinet, D. Hagimont, S. Lacourte et X. Rousset de Pina, ‘‘Is the Micro−Kernel Technology well suited for the support of Object−Oriented Operating Systems: the Guide Experience’’, *Proceedings of the 2nd USENIX Symposium on Microkernels and Other Kernel Architectures (MOKA)*, September 1993.

[Chevalier93b]

P.Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte et X. Rousset de Pina, ‘‘Experience with Shared Object Support in the Guide System’’, *Proceedings of the 4th Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)* , Septembre 1993.

[Chin91]

Roger S. Chin et Samuel T. Chanson, ‘‘Distributed Object−Based Programming Systems’’, *ACM Computing Surveys*, 23(1), pp. 91−124, Mars 1991.

[Daley68]

Robert C. Daley et Jack B. Dennis, ‘‘Virtual Memory, Processes, and Sharing in MULTICS’’, *Communications of the ACM*, 11(4), pp. 308−318, Mai 1968.

[Dasgupta90]

P. Dasgupta, R.C. Chen, S. Menon, M.P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R.J. LeBlanc, W.F. Appelbe, J.M. Bernabeu−Auban, P.W. Hutto, M.Y.A. Khalidi et C.J. Wilkenloh, ‘‘The Design and Implementation of the Clouds Distributed Operating System’’, *Computing Systems*, 3(1), pp. 11−45, Hiver 1990.

[Dasgupta91]

P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra et M.P. Pearson, ‘‘Language and Operating System Support for Distributed Programming in Clouds’’, *Proceedings of the 2nd Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, Mars 1991.

[Day92]

Mark Day, Barbara Liskov, Umesh Maheshwari et Andrew C. Myers, *Naming and Locating Objects in Thor*, Laboratory of Computer Science, MIT, 1992.

[Decouchant91]

D. Decouchant, P. Le Dot, M. Riveill, C. Roisin et X. Rousset de Pina, ‘‘A Synchronization Mechanism for Typed Objects in a Distributed System’’,

Proceedings of the 11th International Conference on Distributed Systems (ICDCS), Mai 1991.

[Freyssinet91a]

A. Freyssinet, *Architecture et Réalisation d'un Système Réparti à Objets*, Thèse de doctorat, Université Joseph Fourier, Juillet 1991.

[Freyssinet91b]

A. Freyssinet, S. Krakowiak et S. Lacourte, ''A Generic Object−Oriented Virtual Machine'', *Proceeding of the 2nd International Workshop on Object Orientation in Operating Systems (IWOOOS)*, Octobre 1991.

[Hagimont92]

D. Hagimont, S. Krakowiak et X. Rousset de Pina, ''Protection in an Object−Oriented Distributed Virtual Machine'', *Proceeding of the 3rd International Workshop on Object Orientation in Operating Systems (IWOOOS)*, Septembre 1992.

[Heiser93]

G. Heiser, K. Elphinstone , S. Russell et G.R. Hellestrand, *A Distributed Single Address−Space Operating System Supporting Persistence*, (9302), School of Computer Science and Engineering, University of New South Wales, Mars 1993.

[Inohara93]

S. Inohara, K. Uehara, H. Miyazawa et T. Masuda, *Sharing Persistent Data Structures on Wide Address Spaces in the Lucas Operating System*, Department of Information Science, Faculty of Science, University of Tokyo, Juillet 1993.

[Jul88]

Eric Jul, *Object Mobility in a Distributed Object−Oriented System*, (88−12−06), Department of Computer Science, University of Washington, Seattle, WA 98195, Décembre 1988.

[Krakowiak90]

S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin et X. Rousset de Pina, ''Design and implementation of an object−oriented strongly typed language for distributed applications'', *Journal of Object−Oriented Programming (JOOP)*, 3(3), pp. 11−22, Octobre 1990.

[Kowalski90]

Oliver C. Kowalski et Hermann Härtig, ''Protection in the BirliX Operating System'', *Proceeding of the 10th International Conference on Distributed Computing Systems (ICDCS)*, pp. 160−166, Mai 1990.

[Lampson71]

B.W. Lampson, ''Protection'', *Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems*, pp. 437−443, Mars 1971.

[Legatheaux88]

J. Legatheaux Martins et Y. Berbers, ‘‘La désignation dans les systèmes d’exploitation répartis’’, *Technique et Science Informatique*, 7(4), pp. 359–372, 1988.

[Levy84]

H.M. Levy, *Capability–Based Computer Systems*, Digital Press, Bedford, 1984.

[Li89]

K. Li et P. Hudak, ‘‘Memory Coherence in Shared Virtual Memory Systems’’, *ACM Transactions on Computer Systems*, 7(4), pp. 321–359, Novembre 1989.

[Liskov85]

B.H. Liskov, *The Argus language and system, Distributed systems: methods and tools for specification*, Lecture Notes in Computer Science, Vol. 190, Springer–Verlag, pp. 343–430, 1985.

[Liskov87]

B.H. Liskov, D. Curtis, P. Johnson et R. Scheifler, ‘‘Implementation of Argus’’, *Proceedings of the 11th ACM Symposium on Operating System Principles, SIGOPS Operating System Review*, 21(5), pp. 111–122, Novembre 1987.

[Liskov92]

B. Liskov, *Preliminary design of the Thor object–oriented database system*, (Programming Methodology Group Memo 74), Laboratory of Computer Science, MIT, 1992.

[Moss90]

J. Eliot B. Moss, ‘‘Design of the Mneme Persistent Object Store’’, *ACM Transactions on Information Systems*, 8(2), pp. 103–139, Avril 1990.

[Mullender86]

S.J. Mullender et A.S. Tananbaum, ‘‘The design of a capability–based distributed operating system’’, *Computer Journal*, 29(8), pp. 289–299, Août 1986.

[Organick72]

E.I. Organick, *The Multics system: an examination of its structure*, MIT Press, 1972.

[Puaut93]

I. Puaut, *Gestion d’objets actifs dans les systèmes distribués : problématique et mise en œuvre*, Thèse de doctorat, Université de Rennes I, Janvier 1993.

[Richardson92]

Joel Richardson, Peter Schwarz et Luis–Felipe Cabrera, ‘‘CACL: Efficient Fine–Grained Protection for Objects’’, *Proceedings of the 7th ACM Conference on Object–Oriented Systems, Languages and Applications (OOPSLA)*, pp. 263–275,

Octobre 1992.

[Rousset93]

X. Rousset de Pina, *Spécification et Construction de Systèmes Opératoires*, Habilitation à diriger des recherches, Université Joseph Fourier, Janvier 1993.

[Rozier88]

M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, P. Léonard, S. Langlois et W. Neuhauser, ''Chorus Distributed Operating Systems'', *Computing Systems*, 1(4), pp. 305–370, Fin 1988.

[Saltzer78]

J.H. Saltzer, *Naming and Binding of objects, in Operating Systems – an advanced course*, Lecture Notes in Computer Science, Vol. 60, Springer–Verlag, pp. 99–208, 1978.

[Schuh90]

Dan Schuh, Michael Carey et David Dewitt, ''Persistence in E Revisited – Implementation Experiences'', *Proceeding of the 4th International Workshop on Persistent Objects Systems*, pp. 345–359, Septembre 1990.

[Stabell93]

Tage Stabell–Kulx et Peter Bosch, *Overview of the Pegasus Storage Architecture*, (rapport interne), Université de Twente, Janvier 1993.

[Verilog93]

ADELE User Manual, Verilog, Février 1993.

[Wulf74]

W.A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson et F. Pollack, ''Hydra: The Kernel of a Multiprocessor Operating System'', *Communications of the ACM*, 17(6), Juin 1974.

Bibliographie complémentaire

[Amaral92]

Paulo Amaral, Rodger Lea et Christian Jacquemot, ''A model for persistent shared memory addressing in distributed systems'', *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pp. 2–12, Septembre 1992.

[Cabrera92]

Luis–Felipe Cabrera, Allen W. Luniewski et James W. Stamos, ''Fine–Grained Access Control in a Transactional Object–Oriented System'', *Computing Systems*, 5(3), pp. 199–216, Eté 1992.

[Garrett92]

William E. Garrett, Ricardo Bianchini, Leonidas Kontothanassis, R. Andrew McCallum, Jeffery Thomas, Robert Wisniewski et Michael L. Scott, *Computer Science Technical Report*, (418), University of Rochester, Avril 1992.

[Koldinger91]

Eric J. Koldinger, Henry M. Levy, Jeffrey S. Chase et Susan J. Eggers, *The Protection Lookaside Buffer: Efficient Protection for Single Address–Space Computers*, (Technical Report 91–11–05), Department of Computer Science and Engineering, University of Washington, Novembre 1991.

[Lea92]

Rodger Lea et Christian Jacquemot, ''The COOL architecture and abstractions for object–oriented distributed systems'', *Proceedings of the 5th ACM SIGOPS European Workshop*, Septembre 1992.

[Luniewski91]

Allen W. Luniewski, James W. Stamos et Luis–Felipe Cabrera, ''A Design for Fine–Grained Access Control in Melampus'', *Proceeding of the 2nd International Workshop on Object–Orientation in Operating Systems (IWOOOS)*, Octobre 1991.

[Moss89]

J. Eliot B. Moss, ''Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach'', *proceedings of the 2nd International Workshop on Database Programming Languages*, Juin 1989.

[Moss92]

J. Eliot B. Moss, ‘‘Working with Persistent Objects: To Swizzle or Not to Swizzle’’, *IEEE Transactions on Software Engineering*, 18(8), pp. 657–673, Août 1992.

[Scott89]

Michael L. Scott, Thomas J. LeBlanc et Brian D. Marsh, ‘‘A Multi–User, Multi–Language Open Operating System’’, *Proceedings of the 2nd Workshop on Workstation Operating Systems (WWOS)*, pp. 115–129, Septembre 1989.

[Scott90]

Michael L. Scott, Thomas J. LeBlanc et Brian D. Marsh, ‘‘Multi–Model Parallel Programming in PSYCHE’’, *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practise of Parallel Programming*, 25(3), pp. 70–78, Mars 1990.

[Scott92]

Michael L. Scott et William Garrett, ‘‘Shared Memory Ought to be Commonplace’’, *Proceedings of the 3rd Workshop on Workstation Operating Systems (WWOS)*, pp. 86–90, Avril 1992.

[Singhal92]

Vivek Singhal, Sheetal Kakkad et Paul Wilson, ‘‘Texas: An Efficient, Portable Persistent Store’’, *Proceedings of the 5th International Workshop on Persistent Object Systems Design Implementation and Use*, Septembre 1992.

[Vaughan92]

Francis Vaughan et Alan Dearle, ‘‘Supporting large persistent stores using conventional hardware’’, *Proceedings of the 5th International Workshop on Persistent Object Systems Design Implementation and Use*, pp. 29–48, Septembre 1992.

[Wilson91]

Paul R. Wilson, ‘‘Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware’’, *Computer Architecture News*, 19(4), pp. 6–13, Juin 1991.

Chapitre II

Objets partagés persistants dans les systèmes répartis : position du problème

Chapitre III

Objets partagés persistants dans les systèmes répartis : différentes approches

Chapitre IV

Expérience du projet Guide : de Guide−1 à Guide−2

Chapitre V

Principes de conception du noyau Eliott

Chapitre VI

Contrôle d'accès dans le noyau Eliott

Chapitre VII

Réalisation du noyau Eliott

Chapitre VIII

Evaluation

Liste des figures

Annexe A

Glossaire du projet Guide

Activité

C’est un flot d’exécution séquentiel à l’intérieur d’un Domaine qui exécute des appels de méthode.

Attribut de visibilité

Un booléen associé à chaque objet indiquant si cet objet peut être appelé depuis un objet appartenant à un autre propriétaire.

Classe

Une classe définit la structure et l’interface d’un ensemble d’objets. Elle est réalisée par un segment contenant une référence externe par méthode vers la librairie de code contenant le code de la méthode.

Couplage d’une grappe

Opération qui rend accessible une grappe dans une tâche par projection dans son espace virtuel.

Domaine

Un espace d’adressage contenant des objets accessibles par des Activités. Un Domaine est créé pour chaque application démarrée et peut être amené à s’exécuter sur plusieurs machines. Un Domaine peut être assimilé à un processus réparti.

Eliott

Le noyau du système Guide−2. Eliott met en œuvre l’environnement d’exécution du système et fournit les fonctions nécessaires aux langages orientés−objets supportés.

Extension

Mécanisme qui permet à une Activité de changer de tâche d’exécution.

Grappe

L’unité de regroupement des objets dans le noyau Eliott. C’est aussi l’unité de couplage dans les tâches composant les Domaines Guide.

Guide

Grenoble Universities Integrated Distributed Environment

Le projet Guide a débuté en 1986 et a pour but la conception et le développement d’un environnement pour le développement d’applications coopératives réparties.

Guide−1

Le premier prototype développé dans le projet Guide. Le Système Guide−1 a été développé sur Unix et sert de support à un unique langage : le langage Guide.

Guide−2

La seconde version du système Guide développée sur le micro−noyau Mach3.0. Elle est composée du noyau Eliott, de services externes et des compilateurs des langages orientés−objets Guide et C++.

Liaison d'une référence

L'opération qui consiste à mettre à jour le vecteur d'accès associé à cette référence externe dans le segment de liaison du segment contenant la référence.

Librairie de code

Un segment contenant des méthodes compilées.

Liste d'accès

Une liste de couple (U, V) associée à un objet et signifiant que l'usager U a le droit d'appeler les méthodes autorisées par la vue V sur cet objet.

Objet

Une instance d'une classe. Elle est réalisée par un segment qui contient une référence vers sa classe.

Propriétaire associé à une tâche

On ne peut coupler dans une tâche que des objets appartenant au même propriétaire. Dans un Domaine, il y a une tâche associée à chaque propriétaire d'objet parmi les objets utilisés.

Propriétaire d'un objet

L'usager du système qui possède cet objet. Lorsqu'un objet est créé, son propriétaire est le propriétaire de l'objet ayant demandé la création.

Segment

Unité d'adressage dans le noyau Eliott. Les objets, les classes et les librairies de code sont réalisés par des segments.

Segment de liaison

Table locale à une tâche, associée à un segment, dont une entrée est associée à chaque référence externe dans ce segment.

Service de stockage

Service du système pouvant être réparti et assurant la persistance des Volumes.

SysRef

Pour *System Reference*. Le nom unique associé à un objet lors de sa création.

Vecteur d’accès

Une entrée dans un segment de liaison. Une telle entrée est associée à une référence externe est donne l’adresse du segment référencé dans la tâche courante et l’adresse de son segment de liaison.

Volume

L’unité d’administration de l’espace de stockage correspondant à la notion de partition de disque.

Vue

Une vue définit dans une classe un ensemble de méthodes autorisées.