
RECHERCHE

Le système réparti à objets Guide

Pierre-Yves Chevalier *
Daniel Hagimont **
Jacques Mossière ***
Xavier Rousset de Pina ***

* ECRC Munich

** INRIA Rhône-Alpes

*** Institut National Polytechnique de Grenoble

Laboratoire IMAG-LSR, BP 53 38041 Grenoble Cedex 9

Internet : Daniel.Hagimont @imag.fr

RÉSUMÉ. Depuis le milieu des années 80, plusieurs équipes de recherche en systèmes se sont intéressées à la gestion d'objets persistants et répartis. Nous rendons compte dans cet article d'une telle expérience, le projet Guide, et, plus précisément, de ses "aspects système" qui permettent une réalisation agréable et efficace d'applications distribuées structurées en termes d'objets persistants. Après avoir décrit les principes de conception et de réalisation du système Guide, nous dressons un bilan de notre expérience, bilan quantitatif sur les performances du système et qualitatif du point de vue de l'écriture des applications. Enfin, nous situons notre travail par rapport aux projets comparables conduits pendant la même période.

ABSTRACT. Since the middle of the 80's, several system research groups have investigated distributed persistent objects management. In this paper, we report on our experiment in designing and implementing such a platform that allows the development of distributed applications in terms of persistent objects. Following a detailed description of this implementation, we provide an evaluation of this research work, regarding both the performance of the system and its adequacy for the development of distributed applications. Finally, we compare our work with related contemporary experiences.

MOTS-CLÉS : systèmes répartis, objets, persistance, adressage, protection.

KEY WORDS: distributed systems, objects, persistence, addressing, protection.

1. Introduction

La programmation orientée objet connaît un succès croissant dans de nombreux domaines, et, en particulier, en génie logiciel et en représentation de connaissances. Le plus souvent, les applications sont conçues en termes d'objets persistants et les

objets sont définis puis créés par l'intermédiaire d'un langage de programmation tel que C++. Enfin, en génie logiciel comme en intelligence artificielle, la tendance est à la construction de systèmes répartis sur des réseaux locaux et, plus récemment, généraux comme Internet. Il est bien sûr possible d'intégrer à chaque nouvelle application une couche logicielle assurant la gestion des objets, de leur persistance et de leur répartition au dessus de systèmes standard. Nous sommes cependant convaincus que le travail des programmeurs et des réalisateurs de compilateurs peut être simplifié sans perte d'efficacité ni de généralité par la réalisation de couches logicielles intermédiaires, communes à toutes les applications.

Nous rendons compte dans cet article de l'expérience d'une réalisation de ce type : le projet Guide, développé de 1986 à 1994, avait pour objectif la réalisation d'une plate-forme système pour des objets persistants et répartis, plate-forme construite au dessus d'environnements standard. Cette plate-forme facilite l'écriture d'applications réparties structurées en termes d'objets. Plus précisément, l'objet est utilisé comme unité de désignation, de partage, de conservation et de protection de l'information. Les applications réparties sont développées à l'aide de langages orientés objet tels que le langage Guide (développé par la même équipe) ou OC++, une extension du langage C++.

Le projet Guide, s'inscrit dans le courant des recherches menées depuis le milieu des années 80 pour étudier, tant du point de vue langage que système, les mécanismes à fournir pour la mise en œuvre efficace d'applications réparties de grandes tailles. Les solutions proposées peuvent être classées en deux catégories :

- Les machines réparties à un seul langage. Le but recherché est d'intégrer dans un langage à objets existant le parallélisme, la synchronisation des accès aux objets, la distribution sur plusieurs sites des objets et même pour certains la persistance. Les mécanismes systèmes nécessaires à la mise en œuvre de ces nouveaux traits du langage sont implantés par son environnement d'exécution et ne sont pas utilisables par un autre langage. Les précurseurs de cette démarche ont été les projets Eden [ALM 85] dont le langage de programmation résultait d'une extension de Concurrent Euclid et Argus [LIS 85] qui étendait le langage CLU pour permettre l'exécution de transactions réparties. Ces projets ont été suivis de nombreux autres : Emerald [BLA 87] (successeur direct d'Eden), ORCA [BAL 92], Gothic-2 [PUA 93]. La première phase du projet Guide, qui a consisté à définir et à mettre en œuvre le langage de programmation à objets persistants partagés et distribués Guide [BAL 91, KRA 90] et son environnement d'exécution sur un réseau de stations de travail Unix, s'inscrit dans cette lignée.

- Les plate-formes distribuées à objets. Le but ici est d'offrir des mécanismes de plus haut niveau sémantique que ceux disponibles sur Unix pour la mise en œuvre de la répartition, du parallélisme et de la persistance tout en restant suffisamment générique pour permettre leur utilisation efficace par les environnements d'exécution de plusieurs langages à objets. Le précurseur des systèmes de la seconde catégorie est le système Clouds [DAS 90] qui fournit une base de mise en œuvre pour les environnements d'exécution de DC++ (une version distribuée de C++) et d'une version distribuée d'Eiffel. Les systèmes SOS [SHA 89], Opal [CHA 92], Cool

[LEA 93], Amadeus [CAH 93b], Corba [SOL 93], Spring [MIT 94] et le système Guide-2 [HAG 94] qui fait l'objet de cet article visent également le même objectif.

Le projet Guide a permis des avancées à la fois sur les plans langage et système. Les principales avancées sur le langage ont été publiées dans [BAL 94]. Nous nous proposons ici de présenter une synthèse des travaux de nature système : en partant d'un modèle d'exécution, nous allons décrire la couche logicielle de Guide permettant de créer des objets persistants, de permettre un accès simultané et contrôlé à ces objets par des processus appartenant à des usagers différents et, enfin, de répartir les objets et les exécutions sur un ensemble de machines reliées par un réseau. Nos expérimentations ont montré qu'il est possible d'implanter un schéma d'adressage et de protection efficace même en l'absence de matériels spécialisés. Ces techniques, à base de segments de liaison, se révèlent meilleures que celles à base de transformations d'adresses (*swizzling*) utilisées au préalable. L'introduction d'une entité de regroupement des objets permet de prendre en compte avec efficacité des objets de tailles très dispersées, variant de quelques dizaines d'octets à quelques centaines de kilooctets.

L'article est articulé comme suit. On trouve en partie 2 une présentation générale du système Guide. En 3 sont présentés les principaux choix de conception du système et une réalisation au dessus du micro-noyau Mach est esquissée en 4. Une évaluation quantitative et qualitative du projet est effectuée en 5, avant de comparer nos résultats aux projets de même nature conduits par d'autres équipes (§ 6). Nous reprenons en conclusion (§ 7) les principaux enseignements du projet ainsi que leur influence sur les travaux que nous menons actuellement.

2. Présentation générale du système Guide

Le système Guide fournit un support d'exécution pour des applications structurées en objets. Les objets, créés dynamiquement comme des instances de classes, sont persistants et répartis sur un ensemble de machines connectées par un réseau. Une application se compose d'un ensemble de processus s'exécutant sur cet ensemble d'objets ; chacun d'eux se compose d'une suite d'appels à des méthodes de ces objets.

2.1. Stockage des objets

Dans Guide, la persistance des objets à l'intérieur des mémoires virtuelles des sites est implicite. Chaque objet créé est persistant, ce qui signifie qu'il survit à la mort du processus l'ayant créé. Certains objets, les racines de persistance, ont des noms prédéfinis et sont directement accessibles (les catalogues, les piles, etc.). Les autres sont accessibles par l'intermédiaire de références figurant dans l'état d'autres objets. Un composant du système, le ramasse-miettes, est chargé de détruire les objets qui ne sont plus accessibles depuis les racines de persistance.

Afin de résister à la panne d'un ou de plusieurs sites, les objets persistants sont recopiés dans une mémoire permanente composée d'un ensemble de disques. Les images sur disques sont utilisées après une panne de site pour reconstruire l'image des objets en mémoire. La cohérence entre l'image d'un objet en mémoire et son image sur disque n'est pas assurée par le système, mais par les applications au moyen d'une demande de recopie explicite sur les disques de l'image en mémoire. Cette opération de recopie permet la mise à jour atomique des images d'un groupe d'objets sur des disques pouvant être différents.

Les objets dont les applications conservent une copie sur disque sont alors dits **permanents**.

2.2. Domaines et activités

Le modèle d'exécution est organisé en espaces d'adressage multi-sites appelés **domaines**. L'opération fondamentale sur les domaines est le couplage qui permet de mettre en correspondance un objet avec un ensemble d'adresses contiguës. Dès qu'un objet est couplé dans un domaine, il est accessible par une adresse. L'ensemble des sites sur lesquels est représenté un domaine et l'ensemble des objets qu'il contient peuvent évoluer dynamiquement. Le couplage permet l'utilisation d'objets préexistant au domaine ou créés par un autre domaine.

Des flots d'exécutions concurrents, que nous appelons **activités**, peuvent s'exécuter dans un même domaine. Pour contrôler les accès simultanés à un même objet par plusieurs activités, des primitives de synchronisation [DEC 91] sont disponibles.

Lors de la création d'un domaine, on y couple un objet initial contenant une méthode de nom prédéfini *main*. On crée également une activité, dite principale, qui exécute cette méthode. L'activité principale peut à son tour créer d'autres activités dans le domaine. L'exécution d'une activité est une suite d'appels de méthodes sur des objets.

2.3. Partage et protection

Un objet peut être couplé, simultanément ou non, dans des domaines différents, ce qui permet le partage d'informations entre domaines.

Les objets étant tous potentiellement partageables, des mécanismes de protection sont nécessaires afin de contrôler les droits d'accès aux objets. Les droits d'accès sont exprimés en fonction de l'utilisateur d'une application ; ils précisent quelles sont les méthodes qu'un programme exécuté pour le compte d'un usager peut appeler sur chaque objet.

La figure 1 donne une vue globale incluant les abstractions définies par le système, à savoir les usagers, les domaines, les activités, les objets partagés, le stockage et la protection.

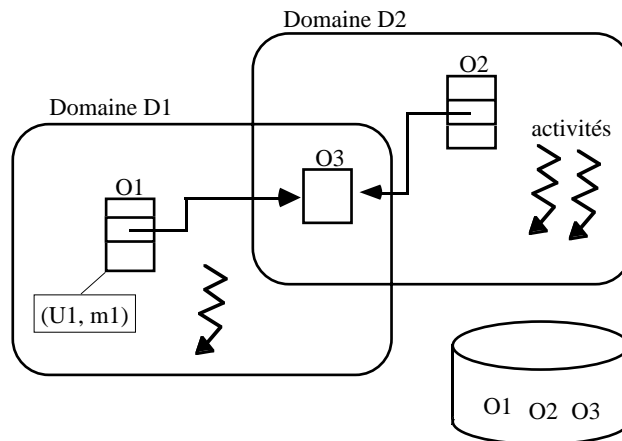


Figure 1. Les abstractions du système Guide. Trois objets O1, O2 et O3 sont définis et ont une copie permanente sur disque. O1 et O3 sont couplés dans le domaine D1, O2 et O3 dans le domaine D2. O3 est donc partagé entre les deux domaines. O1 et O2 font tous deux référence à O3. L'utilisateur U1 a le droit d'appeler la méthode m1 sur l'objet O1.

3. Principes de conception du système

Nous développons dans cette section les idées ayant servi de fil directeur à la conception du système Guide. Nous restons aussi indépendants que possible du système sous-jacent ; nous admettons simplement qu'il permet de définir sur chaque machine un ensemble d'espaces virtuels permettant à des processus d'accéder à des suites d'octets ou segments en leur associant des adresses virtuelles (opération appelée couplage) et qu'il offre des primitives de communication entre les machines reliées au même réseau local.

Nous traitons successivement les problèmes de regroupement, de conservation des objets, d'accès aux objets et enfin de protection.

3.1. Regroupement d'objets

Nos expériences de programmation d'applications ont montré que la plupart des objets gérés sont de petite taille (moins de 300 octets) ; en conséquence, choisir l'objet comme unité d'accès en mémoire de stockage implique le coût d'un couplage lors de chaque liaison d'objet, ainsi qu'une mauvaise utilisation des disques. Nous regroupons donc les objets dans des entités de plus grande taille que nous appelons **grappes**. Le regroupement est effectué sur des critères de localité, de façon à ce qu'environ 80% des références à un objet proviennent d'objets implantés dans la

même grappe. Les grappes ainsi obtenues ont des tailles de quelques dizaines de kilo-octets.

La grappe est l'unité de partage entre les structures d'exécution. C'est aussi l'unité de couplage dans les espaces virtuels composant les domaines. La grappe est une notion qui peut être cachée au programmeur, le système gérant 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.

Un attribut associé à chaque grappe indique si celle-ci peut être couplée sur différentes machines. Lorsqu'une activité tente d'utiliser une grappe déjà couplée sur une autre machine et si le couplage n'est autorisé que sur une machine, l'activité va s'exécuter sur le site où la grappe est couplée ; c'est en particulier le cas des grappes de données modifiables qui sont couplées sur un seul site pour éviter les problèmes de maintien de la cohérence. Dans les autres cas (programmes exécutables et constantes), la grappe est couplée localement.

3.2. *Conservation des objets*

Pour faciliter la gestion de la persistance et le traitement des défaillances, la conservation des objets fait intervenir deux niveaux de mémoire : une mémoire de stockage assimilable en première approximation à un ensemble de disques et une mémoire d'exécution assimilable à l'ensemble des mémoires principales des différentes machines ; chacune de ces mémoires peut être étendue par des zones de disque gérées par des mécanismes de pagination.

Avant toute utilisation, un objet doit être transféré de la mémoire de stockage à la mémoire d'exécution ; il sera recopié en mémoire permanente soit lorsqu'il n'est plus utilisé, soit sur demande explicite effectuée depuis une application. Plus précisément, et pour des raisons d'efficacité, le transfert porte non pas sur un objet unique, mais sur une grappe.

3.2.1. *Stockage*

L'espace de stockage est composé de **volumes** contenant des 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 de 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 l'espace nécessaire à ces grappes est alloué dynamiquement. La figure 2 illustre l'architecture de l'espace de stockage de Guide.

Chaque volume est géré par un **service de stockage**. Un service de stockage est une entité pouvant être répartie. Il assure la permanence des volumes (donc des grappes) et permet la lecture et l'écriture des données du volume lorsque les applications y accèdent. Des exemples de services de stockage sont le système de

fichier d'Unix (avec des montages NFS) ou un ensemble de serveurs dupliqués pouvant être tolérants aux pannes [CHE 94a].

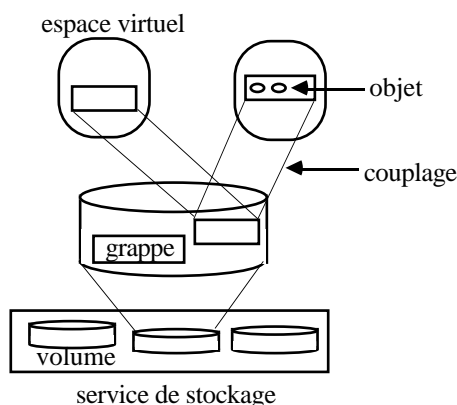


Figure 2. Organisation de l'espace de stockage

Les grappes sont rendues accessibles par couplage en espace virtuel : quand on veut accéder à une grappe, elle est associée à une portion d'espace virtuel ; le chargement des données de ces grappes à partir de l'espace de stockage est réalisé à la demande lors des fautes de pages.

3.2.2. Désignation et localisation

Dans tous les systèmes à objets, on associe à chaque objet un nom interne unique permettant de désigner cet objet sans ambiguïté pendant toute son existence. À partir d'un nom, le système doit être capable de localiser l'objet correspondant. Dans la plupart des cas, le nom unique est complété par des indicateurs permettant d'accélérer le processus de localisation. La longueur des noms d'objets peut alors conduire à une surcharge importante tant en taille de mémoire qu'en temps de transfert entre mémoire et disque ; c'est en particulier le cas dans les systèmes de gestion de bases de données dans lesquelles de nombreux objets contiennent essentiellement des références vers d'autres objets.

Dans le cas de Guide, le système doit être capable de retrouver la grappe de résidence d'un objet à partir de son nom. Le schéma de localisation doit permettre de déplacer un objet d'une grappe à une autre de façon à améliorer dynamiquement la qualité des regroupements. Nous désignons dans la suite un déplacement d'objet sous le terme de migration.

La conception du schéma de désignation et de localisation des objets dans Guide a dû prendre en considération les contraintes suivantes :

- Les noms d'objets doivent garder des tailles raisonnables pour ne pas gaspiller les ressources du système (espace disque et temps du processeur).

- La migration des objets ne doit pas augmenter le coût de la localisation des objets non déplacés (cas général).
- La migration d'objet est essentiellement utilisée pour regrouper les objets dans les grappes afin de rendre plus efficaces les accès futurs. La migration d'un objet doit donc diminuer le temps nécessaire à sa localisation lorsqu'il est dans la même grappe que l'objet qui le localise.

Nous faisons de plus l'hypothèse que les migrations d'objet sont rares.

Les objets sont désignés par une référence système (*SysRef*) de 64 bits, unique dans tout le système. 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 se compose de trois champs : un identificateur de volume (*vol_id*), un identificateur de grappe (*clu_id*) dans ce volume et un identificateur d'objet (*obj_id*) dans cette grappe.

La localisation est alors très rapide pour la grande partie des objets qui ne sont jamais déplacés. La technique utilisée par défaut pour localiser un objet consiste donc à coupler dans l'espace virtuel courant sa grappe de création donnée par sa *SysRef* et à rechercher l'objet dans cette grappe.

La technique utilisée pour gérer la migration d'objet est fondée sur des catalogues de migrations et des liens de poursuite. L'objectif de ces structures de données est d'éviter le couplage de la grappe de création lorsque l'objet a été déplacé dans une grappe déjà couplée. On ramène ainsi le temps de localisation à quelques dizaines de microsecondes alors qu'un couplage aurait demandé une dizaine de millisecondes.

On associe à chaque grappe un catalogue qui enregistre les objets déplacés (arrivés) dans la grappe. Lorsqu'un objet change de grappe de résidence, un lien de poursuite vers l'objet est mis à jour dans sa grappe de création et l'objet est enregistré dans le catalogue de migration de la grappe d'arrivée (ce catalogue fait partie des données de la grappe).

Supposons qu'un objet *O* doive être localisé. Si la grappe de création de *O* (donnée par sa *SysRef*) est déjà couplée dans l'espace courant, alors on recherche l'objet dans cette grappe (c'est le cas par défaut) ; si *O* a été déplacé dans une autre grappe, un lien de poursuite a été laissé et indique la grappe où se trouve l'objet. Si la grappe de création de l'objet n'est pas déjà couplée, alors on cherche dans les catalogues de migration des grappes couplées dans l'espace virtuel courant, afin de vérifier si l'objet n'a pas été déplacé vers une des grappes couplées. Si ce n'est pas le cas, la grappe de création de *O* est couplée et l'objet *O* est recherché dans cette grappe.

Avec cette stratégie, nous ne couplons jamais inutilement une grappe pour un objet se trouvant dans une grappe déjà couplée. De plus, la recherche dans les catalogues de migration des grappes couplées n'a lieu que lorsqu'un couplage de grappe est nécessaire (le coût d'une recherche dans les catalogues de migration est négligeable par rapport au coût d'un couplage).

3.3. Structures d'exécution, partage des objets

Comme nous ne souhaitons pas faire d'hypothèse sur les langages d'écriture des applications, nous ne pouvons pas faire reposer la protection dans Guide sur un contrôle statique effectué à la compilation. En conséquence, nos structures d'exécution et de partage doivent fournir un contrôle dynamique des accès à l'exécution.

Notre objectif a donc été d'assurer l'isolation entre les programmes et les objets, et plus précisément :

- l'isolation entre domaines : une erreur d'adressage dans un domaine D1 ne doit pas provoquer d'erreur dans un domaine D avec lequel D1 ne partage aucun objet.
- l'isolation entre objets : dans un domaine, une erreur dans un objet ne doit pas modifier un autre objet. À défaut de pouvoir mettre en œuvre cette règle de façon stricte, nous nous proposons d'assurer au moins l'isolation entre les objets de propriétaires différents.

Isolation entre domaines

Afin d'assurer l'isolation entre les domaines, ceux-ci ne partagent pas d'espaces virtuels. Chaque domaine est alors implanté par un espace virtuel au moins sur chacun des sites où il est représenté. Le partage d'objets entre les domaines est mis en œuvre en couplant dynamiquement les objets dans un espace virtuel de chaque domaine qui désire y accéder.

Chaque activité de Guide est également réalisée par un ensemble de processus, un par machine. Chacun de ces processus s'exécute dans l'un des espaces virtuels correspondant au domaine sur cette machine. Le processus représentant d'une activité est créé lors du premier appel, et réutilisé lors des appels ultérieurs. L'interaction entre les deux représentants de l'activité prend la forme d'un appel de procédure à distance (RPC) réalisé par échange de messages (figure 3).

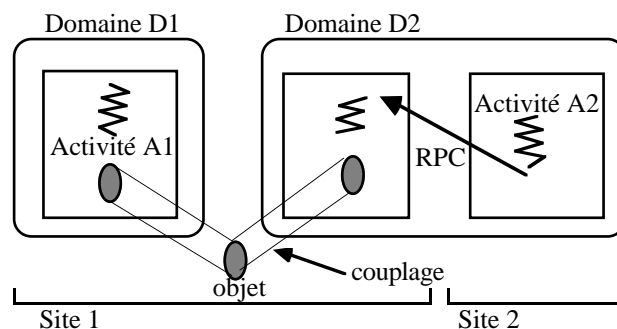


Figure 3. Isolation des domaines. Deux domaines D1 et D2 contiennent chacun une seule activité. D1 étant sur un seul site, il est représenté par un seul espace virtuel, alors que D2 est représenté par un espace virtuel sur chacun des sites 1 et 2.

Un objet est couplé simultanément par D1 et D2 dans les espaces virtuels qui les représentent sur le site 1.

Isolation entre objets

Pour assurer l'isolation entre les objets, il faudrait en toute rigueur disposer dans chaque domaine d'un espace virtuel par objet, ce qui ne serait acceptable que dans le cas de "gros" objets. Dans le cas de Guide, nous avons estimé (et vérifié par des mesures) que ce n'était le cas ni des objets, ni même des grappes. En conséquence, nous interdisons (à l'intérieur d'un domaine) le couplage dans le même espace virtuel d'objets appartenant à des propriétaires différents. Ainsi, une erreur d'adressage malencontreuse dans une méthode ne peut affecter que des objets appartenant au même propriétaire. Un domaine est réalisé sur chaque machine par un ensemble d'espaces virtuels, chacun d'eux correspondant à un propriétaire dont il utilise au moins un objet. Comme l'unité de couplage n'est pas l'objet mais la grappe, ceci implique que tous les objets d'une même grappe appartiennent au même propriétaire.

La figure 4 illustre notre schéma de partage et d'isolation des objets.

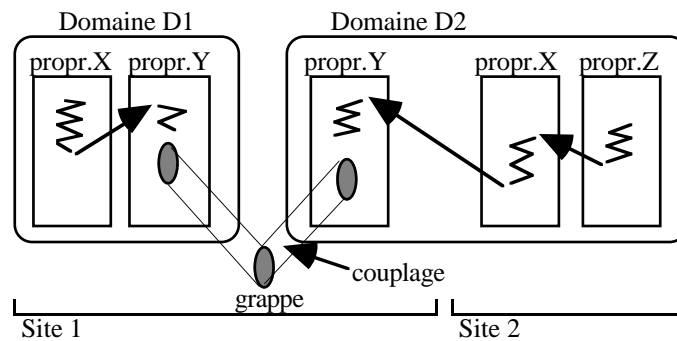


Figure 4. Isolation des domaines et des objets. Comme sur la figure 3, deux domaines D1 et D2 contiennent chacun une seule activité. D1 n'est représenté que sur un seul site, mais il est composé de deux espaces virtuels pour les objets de deux propriétaires (X et Y). D2 est représenté par un espace (pour Y) sur le site 1 et par deux espaces (pour X et Z) sur le site 2. Les deux domaines D1 et D2 partagent sur le site 1 une grappe appartenant à Y.

Un appel de méthode entre des objets appartenant à des propriétaires différents est réalisé par un mécanisme d'appel à distance entre les deux espaces, qu'ils appartiennent au même site ou à des sites différents. Bien que l'isolation au niveau de l'objet ne soit pas strictement appliquée, nous avons constaté que l'isolation par propriétaire constitue un bon compromis entre protection et efficacité.

Modèle à objets

Le système fournit aux compilateurs un modèle d'objets primitif offrant les notions de base qui permettent la construction de modèles d'objets plus complexes. Ce modèle définit trois catégories d'objets : l'**instance**, qui contient les données d'un objet, la **classe**, simple table de pointeurs vers les méthodes, et la **librairie**, qui contient du code exécutable. Les entités correspondantes sont persistantes et sont désignées par des noms internes du système (*SysRef*). La figure 5 montre l'organisation de ces entités.

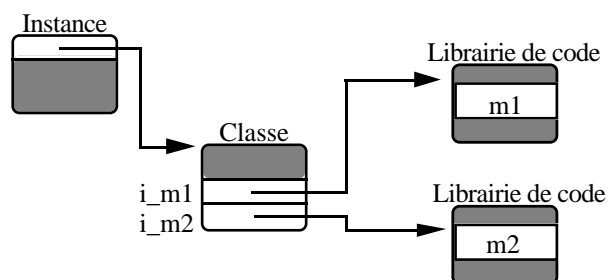


Figure 5. Le modèle à objets primitif

Chaque instance contient une référence à sa classe et ne peut être utilisée que par appel d'une des méthodes définies dans sa classe. Une classe définit l'interface d'utilisation de ses instances et contient une référence externe par méthode, pointant vers la librairie contenant le code de la méthode. Un objet peut contenir une référence à un autre objet. Le code stocké dans les bibliothèques peut faire référence à du code stocké dans d'autres bibliothèques.

Dans la suite, nous ne précisons la catégorie d'un objet (instance, classe, bibliothèques) que lorsque cela est nécessaire.

Adressage et liaison

Pour être utilisé, un objet doit pouvoir être accessible par l'intermédiaire d'une adresse virtuelle. À l'exécution, il faut donc être en mesure de traduire les noms d'objets en adresses virtuelles.

Nous avons rejeté l'approche qui consiste à associer statiquement (à la création) une adresse différente à chaque grappe. Cette approche nécessite de disposer de grands espaces virtuels. Elle ne deviendra réaliste qu'avec la généralisation des processeurs munis d'espaces d'adressage à 64 bits.

Une grappe est donc couplée à des adresses virtuelles différentes dans des espaces virtuels différents et la traduction des noms d'objets est locale à chaque espace. Pour effectuer cette traduction que lors du premier appel à un objet dans un domaine, nous avons utilisé un schéma de liaison analogue à celui de Multics [ORG 72].

Dans chaque espace dans lequel un objet *OI* est couplé, une zone de mémoire est associée à *OI*. Cette zone, appelée **segment de liaison**, est construite à la

alors l'appel de la méthode *m1* sur l'objet désigné par la variable *x* (noté *x.m*) exécutera le code à l'adresse $R[i_x].sl \rightarrow sl[m1].s$ ¹

3.4. Protection

Les objets étant tous potentiellement partageables, il est indispensable de fournir aux applications un moyen de contrôler les droits qu'elles accordent sur les objets qu'elles gèrent. Ces mécanismes doivent répondre aux contraintes suivantes :

- Les droits d'accès doivent s'exprimer en termes de listes des méthodes qu'un programme s'exécutant pour le compte d'un usager a le droit d'appeler.
- Le système doit permettre la délégation de droits. En d'autres termes, il doit être possible d'étendre de manière contrôlée les droits d'un usager pour l'exécution d'une opération spécifique.
- La vérification des droits doit être effectuée dynamiquement : nous ne souhaitons pas faire d'hypothèse sur les compilateurs disponibles, ce qui interdit toute vérification statique.

Nous présentons les mécanismes de contrôle d'accès en fonction des usagers, puis la résolution du problème de délégation.

Contrôle des droits d'accès

Afin de conserver l'efficacité de notre mécanisme d'adressage, nous avons utilisé la technique suivante.

Nous reprenons la notion de **vue**, classique en bases de données. Une vue, définie dans une classe, est un ensemble de méthodes autorisées. Un exemple de définition de vues est le suivant. Considérons par exemple la classe *Fichier* contenant les méthodes *lire* et *écrire*. Il est possible d'y associer 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, c'est à dire l'ensemble des méthodes de l'objet qu'un programme exécuté pour le compte de cet usager a le droit d'appeler. Par exemple, si un objet de classe *Fichier* a la liste d'accès suivante :

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

cela signifie qu'un programme exécuté pour *U1* ne peut qu'appeler *lire* , alors qu'un programme exécuté pour *U2* peut appeler *lire* et *écrire*.

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

¹ Nous utilisons la syntaxe du langage C.

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'utilisateur 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'utilisateur. 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.

Cette solution est illustrée sur la figure 7.

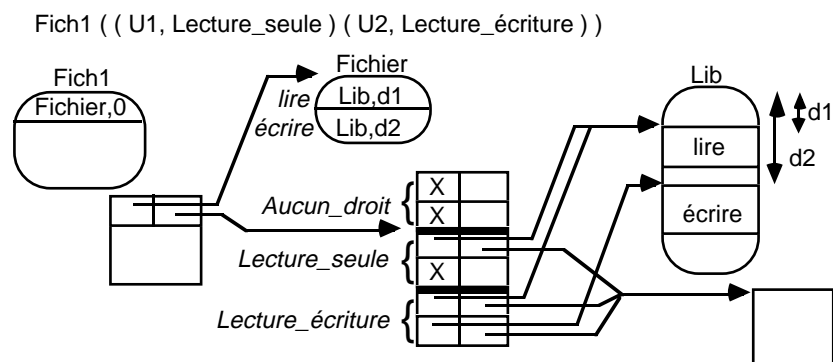


Figure 7. Contrôle par liste d'accès. L'objet *Fich1* de classe *Fichier* est couplé dans un espace virtuel d'un domaine s'exécutant pour le compte de l'utilisateur *U1*. La liste d'accès de l'objet *Fich1* spécifie que *U1* doit utiliser *Fich1* avec la vue *Lecture_seule*. La liaison de la référence 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 engendré pour un appel de méthode est le même qu'auparavant. La protection n'est testée que lors de la liaison de la référence de l'objet vers sa classe et de la classe vers ses méthodes. Une modification dans une liste d'accès ne sera donc prise en compte que lors de la prochaine liaison, mais nous pensons qu'il s'agit d'un compromis acceptable entre les fonctions fournies et l'efficacité de l'adressage.

Une des contraintes énoncées en début de section concerne les garanties offertes par ce mécanisme face à un code erroné ou à une tentative de fraude. L'isolation entre usagers repose sur le fait que des objets de propriétaires différents sont couplés dans des espaces virtuels différents. Le schéma de liaison qui contrôle l'accès aux méthodes est mis en place dans un espace virtuel associé 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 un espace différent. Ainsi, un usager lançant un programme peut seulement atteindre (par manipulation directe d'adresses) ses propres objets ; les

appels de méthodes sur les autres objets passent par la phase de liaison et sont donc protégés.

Délégation des droits

L'objectif de la délégation de droits est d'étendre les droits d'un usager sur un ensemble d'objets à travers un point d'entrée bien défini, ces objets n'étant pas accessibles individuellement. Etant donné que cette fonction est généralement utilisée pour fournir des sous-systèmes protégés de leurs clients, celle-ci doit résister aux tentatives de fraude. La seule protection sûre est la séparation des espaces d'adressage (espaces virtuels).

Lorsqu'un appel de méthode implique un transfert de contrôle entre des objets créés par des usagers différents, cet appel est réalisé par appel de procédure à distance entre deux espaces virtuels et il est interprété. De plus, le propriétaire de l'objet appelant peut être authentifié par le système, puisqu'un propriétaire d'objet est statiquement associé à chaque espace virtuel lors de sa création. Le principe de notre mécanisme est de prendre en compte dans l'expression des droits le propriétaire de l'objet appelant.

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".

Ainsi, lorsqu'un propriétaire X souhaite contrôler l'accès à un ensemble de ses objets, il leur donne l'attribut invisible et crée un objet intermédiaire O , avec l'attribut visible, qui servira d'intermédiaire pour l'accès aux autres objets. Un objet tel que O est appelé un **guichet**.

4. Réalisation

Le but de cette section est de présenter les grandes lignes de la réalisation du système.

4.1. Le micro-noyau Mach

La version actuelle du système Guide a été réalisée sur le micro-noyau Mach [ACC 86]. Sa conception utilise donc les abstractions fournies par ce noyau de système et plus particulièrement :

- les tâches et les processus légers. Une tâche est un espace virtuel dans lequel des segments de mémoire peuvent être rendus accessibles. Cet espace d'adressage est partagé entre les flots de contrôle s'exécutant dans la tâche. Ces flots d'exécution sont appelés processus légers ou *threads*. Un processus Unix est équivalent à une tâche contenant un seul processus léger.

- le couplage de segments. Mach fournit un mécanisme appelé couplage permettant de rendre accessible un segment dans une tâche. Ce segment peut être partagé entre plusieurs tâches.
- les ports et les messages. Un port est une boîte aux lettres permettant de recevoir des messages. Un port appartient à une tâche ; les processus de cette tâche peuvent retirer les messages arrivés sur ce port. L'identité d'un port peut être transmise aux autres tâches donnant ainsi aux processus de ces tâches la possibilité de déposer (envoyer) un message sur ce port.

4.2. Architecture globale

L'architecture globale du système est schématisée sur la figure 8.

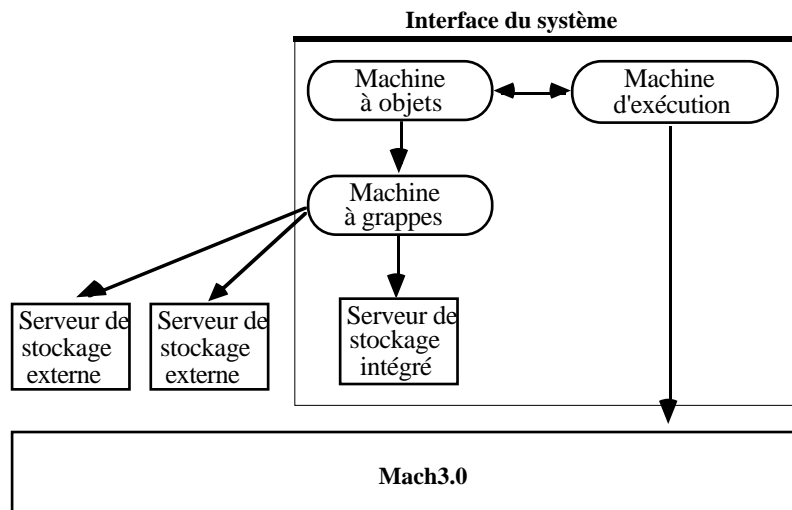


Figure 8. Architecture globale du système

La machine virtuelle du système Guide 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'activités dans un domaine) et à des services relatifs à la gestion des objets (comme la création d'objets ou l'appel de méthode).

Nous appelons **machine à objets** le composant du système qui fournit la gestion d'objets ; les entités gérées sont les instances, les classes et les bibliothèques de code. Cette machine fournit d'une part une interface permettant aux compilateurs de créer des classes et des bibliothèques 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 4.4.

La machine à objets est construite sur la **machine à grappes** présentée en section 4.3. La grappe est l'unité de partage entre les tâches. La machine à grappe fournit essentiellement aux niveaux supérieurs les mécanismes de création, destruction et couplage de grappes. Le stockage des grappes est assuré par des services de stockage. Un service de stockage rapide est intégré au système, mais il est possible d'utiliser des serveurs externes assurant un service de meilleure qualité (haute disponibilité par exemple).

Enfin, la **machine d'exécution** (section 4.5) met en oeuvre les domaines multi-tâches et les activités. Elle fournit également le mécanisme d'appel à distance permettant aux activités de changer de tâche d'exécution.

4.3. Machine à grappes

La gestion des grappes est mise en oeuvre à l'aide de serveurs appelés gérants de mémoire qui sont des paginateurs² de Mach3.0. Il y a en principe un gérant de mémoire par site, bien qu'un site puisse fonctionner avec celui 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. Une opération d'administration du système appelée **montage** permet d'associer un volume à un gérant de mémoire.

Ce gérant de mémoire reçoit alors sous forme de messages les demandes de couplage de grappe émises par les tâches. Les fautes de page sur les grappes sont transmises par Mach au paginateur (gérant de mémoire) qui peut alors contacter le service de stockage pour lire ou écrire les pages demandées ou reçues.

La machine à grappes a également la charge d'assurer l'isolation entre les usagers, étant donné qu'une grappe ne peut être couplée que dans une tâche associée au propriétaire de la grappe. Le gérant de mémoire qui reçoit une demande de couplage doit donc vérifier si ce couplage est autorisé. Si le couplage est interdit, la machine à grappe retourne l'identité du propriétaire de la grappe. La machine à objets peut alors appeler la machine d'exécution pour transférer l'exécution 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 est réalisée par envoi de message au gérant de mémoire en utilisant la notion de port de Mach. Le fait que les ports dans Mach soient protégés par le noyau nous a permis de mettre en oeuvre aisément cette authentification.

Enfin, différents services de stockage offrant différentes qualités de service ont été réalisés pour la gestion des grappes sur disque. Ces services ont été réalisés sur Unix (OSF/1) afin d'éviter la réécriture des pilotes des disques. Il s'agit d'un service de stockage rapide fondé sur le système de fichiers d'Unix et d'un service de stockage fiable gérant les volumes de façon dupliquée.

² Un paginateur est un serveur chargé du traitement des fautes de pages.

4.4. *Machine à objets*

Le but de la machine à objets est de gérer l'allocation des objets et leur utilisation à l'exécution, ces objets pouvant être des instances, des classes ou des bibliothèques de code.

Le principal mécanisme mis en œuvre par la machine à objet est la liaison des références aux objets. Une référence à lier a la forme (*SysRef*, *déplacement*).

Dans chaque tâche, une table appelée *cache d'objets* enregistre toutes les liaisons résolues. Ce cache donne principalement l'adresse de couplage de l'objet et l'adresse de son segment de liaison (le vecteur d'accès à l'objet).

La liaison d'une référence consulte le cache d'objets ; si elle n'y trouve pas l'objet, elle le localise et lui alloue un segment de liaison. Le vecteur d'accès peut alors être placé dans le cache. La liaison est ensuite terminée en ajoutant à l'adresse de couplage de l'objet le déplacement de l'élément référencé. Cette liaison peut intervenir pour une référence à une instance, à une classe ou à une méthode. Nous détaillons ces trois cas.

Lors d'un appel de méthode, l'objet auquel appartient la méthode appelée est désigné par une référence qui peut être stockée dans l'état de l'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. 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 changer de valeur après la liaison. Pour réaliser ce test, chaque objet réalisant une instance contient en en-tête sa propre *SysRef*, ce qui permet de détecter si le 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. En effet, un mécanisme de liaison anticipée est mis en œuvre par la machine à objets, puisque l'utilisation d'un objet implique l'utilisation de sa classe.

Enfin, la liaison de la référence de la classe appelée au code de la méthode appelée est testée et provoque le cas échéant l'appel de la primitive système qui effectue la liaison.

Toutes ces liaisons prennent en compte la définition de la protection comme cela est décrit en section 3.3.

4.5. *Machine d'exécution*

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

Les domaines sont réalisés par des tâches Mach et les activités par des processus légers dans ces tâches. Etant donné qu'il n'était pas possible avec la version du

micro-noyau Mach utilisée de créer des tâches ou des processus légers à distance (à partir d'une autre machine que celle de création), un démon s'exécute sur chaque site et reçoit les requêtes de création des autres sites.

Pour permettre à une activité de changer de tâche, un processus léger lui est associé dans chaque tâche où elle s'est exécutée. Un port est associé à ce processus. Un changement de tâche est alors réalisé par l'envoi à ce port d'un message correspondant à une requête d'exécution. 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 (à travers une table gérée dans chaque tâche), il peut changer de tâche par un simple appel de procédure à distance. 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.

5. Evaluation

Dans cette section, nous donnons tout d'abord une vision globale des expérimentations menées dans le projet Guide, à savoir le développement de prototypes et d'applications réparties. Puis nous présentons une évaluation quantitative du prototype final fondée sur les mesures que nous avons pu faire avec certaines applications. Enfin nous présentons une évaluation qualitative dans laquelle nous reconsidérons certaines hypothèses faites au début du projet.

5.1. Expérimentations

Le système Guide a fait l'objet de deux mises en œuvre l'une sur Unix (Guide-1) et l'autre sur le micro-noyau Mach 3.0 (Guide-2). Ces deux mises en œuvre ont été réalisées en C et représentent un volume de code sensiblement identique d'environ 50000 lignes.

Le premier prototype, grâce à la grande diffusion d'Unix, a été utilisé pour la réalisation de trois applications de taille significative : l'éditeur structuré coopératif Griffon [DEC 93] développé au sein de notre laboratoire, le service d'annuaire X400 développé à Bull et une application de circulation de documents administratifs développée au SEPT à Caen [CAH 93a]. Les principales critiques concernant le premier prototype avaient trait, d'une part, à ses lacunes : absence de mécanismes de protection et non prise en compte des pannes et, d'autre part, au manque d'efficacité des mécanismes d'accès aux objets (couplage, chargement, liaison et appel de méthode).

Le second prototype, celui que nous avons présenté dans cet article, tente de remédier à ces inconvénients. Nous nous intéressons dans la suite (5.2) à l'évaluation des différents mécanismes d'accès aux objets qui sont réalisés pour partie par la machine à grappe et pour partie par la machine à objets. Cette évaluation est fondée sur l'observation du comportement de ces deux machines lors de l'exécution

de plusieurs programmes de test. Il s'agit d'un tableur réparti multi-utilisateur [CHE 94b], des tours de Hanoï (une application récursive), de l'éditeur coopératif de document Griffon [DEC 93] et d'un banc d'essai appelé le "Cattell Benchmark" [CAT 92].

5.2. Evaluations quantitatives de Guide-2

Comme cela a été souligné précédemment, un objectif important du second prototype est l'efficacité des mécanismes d'accès aux objets implantés dans la machine à grappes et la machine à objets.

5.2.1. Evaluation de la machine à grappes

Afin de minimiser le nombre d'opérations de couplage, les objets sont regroupés dans des grappes. L'efficacité de ce mécanisme est donc très liée à l'efficacité de la politique de regroupement des objets dans des grappes. En effet, si une grappe regroupe des objets logiquement liés, alors le couplage d'un objet provoquera le couplage de son espace de travail, c'est à dire des objets qui ont une forte probabilité d'être référencés au cours de la même exécution.

Cette constatation militerait donc pour la définition de grappes très grandes et, à la limite, contenant l'ensemble des objets et du code d'une application. Cependant, la grappe étant l'unité de partage, sa taille doit rester raisonnable, ceci pour des raisons de protection en ce qui concerne les grappes de données (les grappes de code peuvent être couplées dans tous les contextes) et, plus généralement, de taille de l'espace virtuel disponible.

En outre, à un niveau plus fin, en regroupant dans la même page plusieurs objets, la lecture d'un objet entraînera celle de ces voisins ce qui réduit le nombre d'accès au disque et donc le coût du chargement d'un objet en mémoire. Dans la mesure où les opérations d'entrées sorties sur disques restent coûteuses, cela améliorera les performances globales du système.

Nous avons observé l'utilisation des grappes par trois des applications de démonstration de notre système : le tableur réparti multi-utilisateur, les tours de Hanoï et l'éditeur coopératif de document Griffon. Le nombre de grappes utilisées par ces applications est très important relativement à ce que nous envisagions. Par exemple, l'éditeur coopératif Griffon utilise 27 grappes de codes et 56 grappes de données pour la création de 5 documents au cours d'une session de travail regroupant trois usagers. Le nombre élevé de grappes de code montre que le créateur d'application, en l'absence de directive, crée pour chacune des classes qu'il utilise une grappe différente pour y ranger son code. Le nombre de grappes de données, environ 3 grappes par document et par utilisateur, résulte des contraintes de protection. Dans la mesure où l'implantation des données dans des grappes est essentiellement dirigée par la politique de protection définie par l'application, on ne peut que le constater et l'utilisateur paie en terme de performance la sécurité qu'il demande. En revanche, le choix de politiques de regroupement du code des classes est libre de toute contrainte

et, à la limite, une application n'utilisant que les classes qu'elle a définies ne devrait utiliser qu'une grappe de code. Une politique efficace de regroupement du code ne peut être déterminée qu'après une phase d'observation de l'application.

Pour terminer cette courte évaluation de la machine à grappes, dont une étude plus approfondie peut-être trouvée dans [CHE 94b], nous donnons le temps d'exécution de ses principales primitives. Ces temps ont été mesurés sur des machines Zénith i486 possédant une fréquence d'horloge de 33 MHz avec 24 Mo de mémoire centrale (dont 12 à 14 Mo sont disponibles pour les applications) et sur lesquelles s'exécutent un noyau Mach 3.0 (Norma MK14) et le système OSF/1 MK (V4.1). Les disques utilisés sont des disques IDE dont le débit maximal en lecture est de 860 Ko/s.

Machine à Grappes	temps (en ms)
création d'une grappe de 82 Ko	186, 0
couplage d'une grappe	8, 0
lecture d'une page depuis le stockage	3, 8

La création d'une grappe est une opération atomique et un quart environ de son coût provient des opérations qui assurent cette propriété. Mais, même si elle n'avait pas été rendue atomique, la création d'une grappe serait coûteuse car elle demande l'allocation d'une référence unique, d'un descripteur de grappe ainsi que la création d'un fichier Unix qui représente la grappe en mémoire de stockage.

Le couplage d'une grappe implique la lecture préalable de son descripteur afin de connaître sa taille et son propriétaire. Cependant cette lecture, comme celle du reste d'une page, bénéficie du cache de pages ainsi que de la politique de lecture en avant associée au système de fichiers Unix.

Les opérations sur les grappes sont coûteuses, mais ces coûts sont amortis lorsque les objets d'une même grappe sont fréquemment utilisés.

5.2.2. Evaluation de la machine à objets

Le principal mécanisme mis en œuvre par la machine à objets est le mécanisme de liaison dynamique. Nous avons mesuré le coût de ce mécanisme dans différents cas de figure, puis, afin de connaître l'efficacité du mécanisme, nous avons étudié sur quelques applications le nombre de réutilisations par une application d'une référence déjà liée.

Les mesures ci-dessous donnent le temps nécessaire à l'appel d'une méthode sur un objet dans différents cas de figure. Ces mesures ont été effectuées sur les machines Bull-Zenith P.C. 486 décrites auparavant :

(1) appel de méthode sans défaut	4, 4 μ s
(2) appel de methode avec liaison	entre 22 μ s et 55 μ s

La mesure (1) correspond à l'appel d'un objet déjà lié qui s'exécute donc sans appel aux primitives du système. Ce temps doit être comparé à l'appel de fonction en C sur le même processeur (0, 9 μ s) et à l'appel d'une méthode virtuelle en C++ (1, 5 μ s). Le surcoût par rapport à C++ représente le coût d'empilement des paramètres supplémentaires rajoutés par notre mécanisme ainsi que le coût de la vérification de la validité de la liaison.

La borne inférieure de la mesure (2) représente le cas où l'objet n'est pas lié mais est dans le cache d'objet de la tâche courante. On peut donc estimer que dans ce cas la liaison proprement dite de l'objet coûte 17, 6 μ s. La borne supérieure représente le cas où l'objet n'est pas dans le cache, mais où la grappe qui le contient est déjà couplée. Les 55 μ s représentent donc le coût de localisation de la grappe, la mise à jour du cache d'objets et la liaison. Dans le cas où la grappe n'est pas couplée, le coût du maintien des caches et de la liaison est négligeable devant le coût des opérations nécessaires (faute de page si la grappe peut être couplée dans le contexte de l'appelant ou le coût de l'appel à distance sinon). Des mesures détaillées des coûts de l'appel à distance dans les différents cas de figure (selon qu'il faut ou non étendre le domaine ou étendre l'activité) sont présentés dans [BAL 93].

Nous avons également mesuré le pourcentage des appels utilisant une référence déjà liée pour trois applications : le tableur réparti multi-utilisateur, les tours de Hanoï, et le "Cattell benchmark", une application très utilisée dans la communauté des bases de données. Les pourcentages mesurés sont les suivants :

tableur réparti multi-utilisateur	85 %
tours de Hanoï	37 %
Cattell benchmark avec 1000 nœuds	91 %
Cattell benchmark avec 5000 nœuds	71 %
Cattell benchmark avec 8000 nœuds	58 %

Les mesures faites sur le tableur montrent que le taux de réutilisation des références déjà liées est très important, au moins dans le cas d'une application ne comportant que peu de classes. Cependant, les mesures faites sur l'application suivante montrent que ceci n'est pas vrai pour les applications qui, comme les tours de Hanoï, accèdent aux objets par des références passées en paramètres et donc localisées sur la pile de l'activité (cas des applications ayant un fort taux de récursivité). La résolution de la référence est donc perdue chaque fois que l'on sort de la méthode. Cependant, pour ces applications, le taux de réutilisation reste suffisamment intéressant pour justifier notre mécanisme.

Le Cattell benchmark met en œuvre un cheminement dans un graphe. Ce graphe est constitué de nœuds dont chacun possède trois fils choisis au hasard parmi le nombre total de nœuds qui est un paramètre de l'application. Le cheminement est fait à partir d'un nœud jusqu'à sept niveaux de profondeur, en appelant une méthode vide sur chacun des objets visités (3280 appels). Les pourcentages mesurés dans le cas de cette application reflètent le fait que les chances de réutiliser une référence déjà liées dépendent fortement de la probabilité pour deux objets visités de contenir une

référence sur le même objet. Cette probabilité diminue au fur et à mesure que le nombre d'objets croît. Mais cette application justifie encore le choix de notre mécanisme de liaison.

5.3. Evaluations qualitatives

Après cette évaluation de la machine à grappes et de la machine à objets, nous revenons sur certaines hypothèses formulées au début ou au cours du projet et qui méritent d'être reconsidérées à présent.

5.3.1. Support multi-langage

Dans la seconde réalisation de Guide, nous avons voulu fournir une machine virtuelle générique devant répondre aux besoins des compilateurs d'une grande classe de langages orientés objets. La machine que nous avons conçue a permis le support de deux compilateurs pour les langages Guide et OC++.

Malheureusement, le modèle à objet que nous fournissons aux réalisateurs de compilateurs, s'il est adapté au support de l'héritage simple, ne permet pas une réalisation efficace de l'héritage multiple. Ceci milite pour une gestion d'objets adaptée à chaque compilateur et s'appuyant sur la machine à grappe.

5.3.2. Généricité de la machine à grappe

Durant le projet Guide, nous avons également coopéré avec une équipe de base de données réalisant un système de gestion de base de données orienté objet. Nous avons observé que, si la gestion des objets différait fortement de la notre, la gestion des données au niveau de la machine à grappes était semblable.

Il semble donc que la généricité que nous recherchons se situe plutôt au niveau de la machine à grappes qu'à celui de la machine à objets.

5.3.3. Adressage

Une hypothèse forte du projet a été que nous ne disposions pas d'espaces virtuels suffisamment grands pour contenir tous les objets du système. Cette hypothèse qui était vérifiée en 1990 est actuellement remise en cause avec l'arrivée des microprocesseurs à 64 bits d'adresse virtuelle. Des projets comme Opal [CHA 92] (voir section suivante) étudient déjà cette approche, l'avantage principal étant de désigner directement chaque objet par son adresse.

6. Comparaisons avec des projets analogues

Comme nous l'avons mentionné dans l'introduction, fournir des outils pour la mise en œuvre d'applications réparties à base d'objets a conduit à deux types de projets : ceux visant à fournir distribution, parallélisme, synchronisation et

persistance comme de nouveaux traits intégrés à un langage de programmation ; ceux qui fournissent ces nouvelles fonctions comme des extensions à un système existant. Le système Guide-2 a contribué à l'effort de recherche mené par les différents projets du second type. Nous présentons dans la suite de cette section ce que nous pensons être notre contribution.

Le principal but du projet Clouds est la construction d'un environnement de calcul résistant aux fautes et qui apparaisse à ses utilisateurs comme une ressource uniforme et intégrée. Le système est basé sur la notion d'objets passifs, d'appel de méthode indépendant de la localisation de l'objet cible et d'actions (transactions) imbriquées. Un objet au sens de Clouds est une mémoire virtuelle persistante composée de code, de données et de points d'entrée. Les entités actives sont des flots de contrôle qui démarrent dans un objet et se déplacent d'un objet dans un autre grâce au mécanisme d'appel de méthode. Contrairement à Guide l'appel de méthode local est distinct de l'appel de méthode à distance. C'est donc toujours le programmeur qui décide si l'exécution de l'appel se fera localement ou à distance. Le partage d'objets entre des flots de contrôle s'exécutant sur des sites différents est réalisé par copies multiples gérées en cohérence stricte au niveau de la page. Les mécanismes fournis par Clouds s'adressent donc à des objets à très gros grains et supposent que ce seront les compilateurs qui devront prendre en charge complètement la gestion des accès (synchronisation, adressage et protection) aux objets de petite taille.

Le projet SOS est une plate-forme distribuée à objets basée sur le principe du mandataire (proxy) : l'interface d'un service, mise en œuvre par un ensemble d'objets coopérants, est un objet de communication, le mandataire de ce service. L'unité d'allocation, de stockage et de communication est l'objet qui peut être de petite taille (une centaine d'octets). Le système n'implante que les mécanismes de base pour appeler une méthode, déplacer ou détruire un objet. Les autres fonctions du système (localisation, communication, stockage et désignation) sont fournies comme des services. Le modèle d'exécution fourni est le modèle client-serveur. Un objet ne peut être chargé que dans l'espace virtuel d'un seul processus qui joue alors le rôle de serveur pour cet objet. Hormis le fait que le projet SOS est très orienté vers C++, sa différence essentielle avec Guide-2 est qu'il ne cherche pas à intégrer les mécanismes qu'il fournit pour l'écriture d'applications réparties.

Cool v2 et Amadeus sont deux plate-formes qui ont, comme Guide-2, bénéficié des travaux menés au sein des deux projets Esprit Comandos [CAH 93a]. La différence essentielle entre ces deux projets et Guide-2 est la technique qui a été choisie pour "supporter" les langages à objets. Cool et Amadeus, contrairement à Guide, n'exportent pas de modèle d'objet vers les langages. Ils fournissent pour coopérer avec les environnements d'exécution des différents langages un mécanisme d'appel ascendant (*upcall*) qui permet au système d'appeler des fonctions exportées par le compilateur. Ce mécanisme permet de conserver les schémas d'adressage et d'exécution des compilateurs existants et en limite donc les modifications au prix d'une plus grande lourdeur des mécanismes systèmes. L'adressage dans ces deux projets est fondé sur la technique de mutation de pointeur (*swizzling*). Chaque nom d'objet est remplacé à l'exécution par une adresse virtuelle lors de la première

utilisation. Toutefois, si un objet ne peut pas être couplé à la même adresse dans tous les processus, tous les couplages de l'objet sont invalidés, puis reconstruits à la demande.

Plus récemment, le système Opal propose d'exploiter les microprocesseurs à espace d'adressage 64 bits pour gérer un espace virtuel unique. La machine virtuelle fournie par Opal est une machine segmentée qui peut-être comparée à la machine à grappes fournie par Guide-2. Tout segment (ensemble de pages contiguës) se voit allouer à sa création une adresse virtuelle unique qui est à la fois son nom et l'adresse à laquelle il sera toujours utilisé, ce qui augmente l'efficacité des mécanismes d'adressage. Un mécanisme de protection basé sur des capacités logicielles et des domaines de protection permet de définir des droits différents sur l'espace unique de segments. Cependant, les mécanismes fournis sont à gros grain (le segment) et ce sont donc les compilateurs qui devront engendrer le code contrôlant les accès aux entités logiques (objets) regroupées dans les segments.

Corba (Common Object Request Broker Architecture) est une spécification d'interface proposée par un consortium (Object Management Group ou OMG) regroupant des constructeurs et des fabricants de logiciel. Corba vise à permettre à des applications en provenance de constructeurs différents de coopérer en les encapsulant dans des objets dont l'interface est celle de l'application (API). Corba spécifie un appel de méthode à distance et un langage de description d'interface (IDL) qui permet la coopération entre des modèles d'objets différents. Le modèle d'exécution de Corba est un modèle client serveur qui est très contraignant dès lors que l'on veut définir des serveurs à plusieurs flots de contrôle. D'autre part Corba est difficilement utilisable sans les services objets (*Object services*) dont la spécification n'est pas, à ce jour, complète. Quelques implantations de Corba sont disponibles, dont celle de Sun appelée ODE et celle d'Iona technologies appelée Orbix [ORB 93].

Spring est un système distribué à objets développé chez Sun Microsystems. Spring inclut un micro-noyau et un environnement d'exécution distribué similaire à Corba (c'est à dire un langage de description d'interface et des services objets). Le micro-noyau fournit des domaines, des flots de contrôles et des portes ; il gère également la mémoire virtuelle. Les domaines et les flots sont analogues aux domaines et aux activités de Guide. Les portes sont des objets de communication protégés attachés à un domaine et qui représentent des droits d'appel à des points d'entrée d'autres domaines. Les portes sont utilisées également pour augmenter l'efficacité des appels inter-domaines locaux à un site. Les appels inter-domaines sur des sites différents utilisent des représentants à distance. Le langage de description d'interface est celui de l'OMG. Spring fournit deux catégories d'objets : ceux qui sont toujours manipulés via un serveur et ceux qui sont manipulés dans l'espace des clients. Pour la première catégorie, l'IDL construit des talons qui réalisent l'appel au serveur via les mécanismes de portes et de représentant. Pour les seconds, un mécanisme de recopie dans l'espace des utilisateurs est fourni. Les services objets implantés au dessus du micro-noyau comprennent entre autres un gestionnaire des représentants, le serveur de noms, la gestion des fichiers, et un éditeur de liens dynamique.

7. Conclusion

Pour conclure, nous résumons les enseignements tirés de notre expérience, puis nous présentons les perspectives que nous offre ce travail.

7.1. Enseignements

Les expérimentations conduites dans le cadre du projet Guide ont prouvé la validité de nos convictions initiales, tant sur le plan de la commodité d'écriture des applications que sur celui de l'efficacité d'exécution. Nous estimons cependant que notre réalisation rencontre trois limites principales.

En premier lieu, le choix de Mach comme support se révèle peu judicieux compte tenu de l'abandon de fait de ce micro-noyau par les principaux constructeurs. Nous avons également surestimé les problèmes d'efficacité du prototype construit au dessus d'Unix.

En second lieu, le couplage des grappes et les opérations de liaison restent coûteuses, bien qu'elles soient inévitables dans un contexte où les espaces virtuels sont trop petits pour contenir en permanence l'ensemble des objets du système.

Enfin et surtout, nous avons réalisé un support de trop haut niveau pour les objets : la machine Guide manque de généricité pour permettre de réaliser efficacement aussi bien des systèmes de bases de données que des applications coopératives. Il aurait été plus souple de construire une interface au niveau de la machine à grappes et de laisser la gestion des objets à la charge des compilateurs.

En conséquence, notre projet de recherche actuel cherche à éviter ces trois obstacles.

7.2. Perspectives

Nous nous proposons de construire au dessus d'un système Unix standard une mémoire virtuelle répartie utilisant les capacités d'adressage à 64 bits. Cette mémoire répartie contient des segments, ensembles d'octets contigus persistants et attachés pour toute leur vie à des adresses virtuelles fixes. Les segments sont protégés par des mécanismes à base de capacité et de domaines. Par rapport aux réalisations comme Opal [CHA 92], l'originalité la plus marquante est de laisser à l'utilisateur le choix de ses politiques de cohérence, la cohérence pouvant être exprimée en termes de zones de longueur quelconque.

Ce projet, appelé Sirac, est spécifié et un premier prototype est en cours de réalisation.

8. Remerciements

Le projet Guide dont nous avons décrit la seconde phase a bénéficié du soutien de la communauté européenne dans le cadre des deux projets Esprit Comandos 1 (Esprit 834) et Comandos 2 (Esprit 2071). Ce projet n'aurait pas pu être mené à son terme sans la participation active de R. Balter, J. Bernadat, J. Cayuela, D. Decouchant, A. Duda, A. Freyssinet, H. Jamrozik, S. Krakowiak, S. Lacourte, M. Meysembourg, H. Nguyen Van, P. Le Dot, M. Riveill, C. Roisin, M. Santana, R. Scioville et G. Vandôme. Nous remercions également les "relecteurs" de TSI pour leurs nombreux conseils.

9. Bibliographie

- [ACC 86] M.J. ACETTA, R. BARON, W. BOLOWSKY, D. GOLUB, R. RASHID, A. TEVANI, M. YOUNG, "Mach: a New Kernel Foundation for Unix Development", *USENIX 1986 Summer Conference*, p. 93-112, Juillet 1986.
- [ALM 85] G. T. ALMES, A. P. BLACK, E. D. LAZOWSKA, AND J. D. NOE, "The Eden System: A Technical Review", *IEEE Transactions on Software Engineering.*, Vol 11(1), p. 43-59, Janvier 1985.
- [BAL 92] 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), p. 190-205, Mars 1992.
- [BAL 91] 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), p. 31-67, Hiver 1991.
- [BAL 93] R. BALTER, P.Y. CHEVALIER, A. FREYSSINET, D. HAGIMONT, S. LACOURTE, X. ROUSSET DE PINA, "Is the Micro-Kernel Technology well suited for the support of Object-Oriented Operating Systems: the Guide Experience", *2nd Symposium on Microkernels and Other Kernel Architectures (MOKA)*, San Diego, p. 1-11, Septembre 1993.
- [BAL 94] R. BALTER, S. LACOURTE, M. RIVEILL, "The Guide Language: Design and Experience", *Computer Journal*, 37(6), p. 519-530, Décembre 1994.
- [BLA 87] A. BLACK, N. HUTCHINSON, E. JUL, H. LEVY, ET L. CARTER, "Distribution and Abstract Type in Emerald", *IEEE Transaction on Software Engineering*, 13(1), p. 65-76, janvier 1987
- [CAH 93a] V. CAHILL, R. BALTER, X. ROUSSET DE PINA, N. HARRIS, *The Comandos Distributed Application Platform*, Springer-Verlag, p. 123-140, 1993.
- [CAH 93b] V. CAHILL, S. BAKER, C. HORN, G. STAROVIC, "The Amadeus GRT - Generic Runtime Support for Distributed Persistent Programming", *8st ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, p. 144-161, Octobre 1993.

- [CAT 92] R. CATTELL, J. SKEEN, "Object Operation benchmark, *ACM Transactions on Database Systems*, 17(1), p. 1-31, Mars 1992.
- [CHA 92] J. S. CHASE, H. M. LEVY, E. D. LAZOWSKA, M. BAKER-HARVEY, "Lightweight Shared Objects in a 64-Bit Operating System", *7th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 27(10), p. 397-413, Octobre 1992.
- [CHE 94a] P.Y. CHEVALIER, *Persistence et Disponibilité dans les Systèmes Répartis*, Thèse de Doctorat en Informatique, Université Joseph Fourier (Grenoble I), Octobre 1994.
- [CHE 94b] P.Y. CHEVALIER, M. RIVEILL, F. SAUNIER, "Toward a Generic System Support for Co-operative Application", *3rd Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprise*, IEEE Computer Society Press, West Virginia, p. 1-10, Avril 1994.
- [DAS 90] 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, C.J. WILKENLOH, "The Design and Implementation of the Clouds Distributed Operating System", *Computing Systems*, 3(1), p. 11-45, 1990.
- [DEC 91] D. DECOUCHANT, P. LE DOT, M. RIVEILL, C. ROISIN, X. ROUSSET DE PINA, "A Synchronization Mechanism for Typed Objects in a Distributed System", *11th International Conference on Distributed Systems (ICDCS)*, p. 152-161, Septembre 1991.
- [DEC 93] D. DECOUCHANT, V. QUINT, M. RIVEILL, I. VATTON, *Griffon: A Cooperative, Structured, Distributed Document Editor*, (93-01), Bull-IMAG, Mai 1993.
- [HAG 94] D. HAGIMONT, P.Y. CHEVALIER, A. FREYSSINET, S. KRAKOWIAK, S. LACOURTE, J. MOSSIÈRE ET X. ROUSSET DE PINA, "Persistent Shared Object Support in The Guide System: Evaluation & Related Work", *9th ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, p. 129-144, Octobre 1994.
- [KRA 90] S. KRAKOWIAK, M. MEYSEMBOURG, H. NGUYEN VAN, M. RIVEILL, C. ROISIN, 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), p. 11-22, Octobre 1990.
- [LEA 93] R. LEA, C. JACQUEMOT ET E. PILLEVESSE, "COOL: system support for distributed object-oriented programming", *Communications of the ACM, Special issue on Concurrent Object Oriented Programming*, 36(9), p. 37-46, Septembre 1993.
- [LIS 85] B.H. LISKOV, The Argus language and system, *Lecture Notes in Computer Science (Distributed systems: methods and tools for specification)*, Vol. 190, Springer-Verlag, p. 343-430, 1985.
- [MIT 94] J. MITCHELL, J. GIBBONS, G. HAMILTON, P. KESSLER, Y. KHALIDI, P. KOUGIOURIS, P. MADANY, M. NELSON, M. POWELL AND S. RADIA, "An Overview of the Spring System", in A Spring Collection: a collection of papers on the Spring Distributed O-O Operating System, SunSoft, Septembre 1994.
- [ORB 93] *Orbix - a technical overview*, Iona Technologies, The O' Reilly Institute, Westland Row, Dublin 2, Ireland, July 1993.

- [ORG 72] E.I. ORGANICK, *The Multics System: an Examination of its Structure*, MIT Press, 1972.
- [PUA 93] I. PUAUT, *Gestion d'objets actifs dans les systèmes distribués : problématique et mise en œuvre*, Thèse de doctorat, Université de Rennes I, Janvier 1993.
- [SAN 93] M. SANTANA, *Manuel de Référence du Langage OC++*, Centre de Recherche Bull, 2 rue de Vignate Gières 38, Décembre 1993.
- [SHA 89] M. SHAPIRO, Y. GOURHANT, S. HABERT, L. MOSSERI, M. RUFFIN, C. VALOT, "SOS: An Object-Oriented Operating System - Assessment and Perspectives", *Computing Systems*, 2(4), p. 287-337, 1989.
- [SOL 93] R. M. SOLEY, *Object Management Architecture Guide*, Rev. 2.0, 2nd edition, Wiley-QED, 1993