Un Bilan du système réparti à objets SOS

Marc Shapiro, Yvon Gourhant, Sabine Habert, Jean-Pierre Le Narzul,
Laurence Mosseri, Michel Ruffin, Celine Valot

Un bilan du système réparti à objets SOS*

An Assessment of the Object-Oriented Distributed System SOS

Marc Shapiro, Yvon Gourhant, Sabine Habert,
Jean-Pierre Le Narzul, Laurence Mosseri,
Michel Ruffin, Céline Valot

Institut National de Recherche en Informatique et en Automatique
INRIA, B.P. 105, 78153 Le Chesnay Cédex, France
tel.: +33 (1) 39-63-55-11, télex: 697 033 F
e-mail: sos@sor.inria.fr

11 mai 1990

*Ce travail a été financé en partie par la Commission de la Communauté Européenne, sous contrat Esprit 367 SOMIW.

Résumé

Nous présentons ici un système à objets réparti appelé SOS. SOS a pour but de faciliter la programmation d'applications réparties, grâce au mécanisme des objets fragmentés. SOS offre de plus des mécanismes génériques pour la gestion des objets composant les applications : invocation, identification, migration, stockage. Nous présentons le modèle d'objets de SOS, composé d'objets élémentaires et fragmentés. Ce modèle garantit l'intégrité de type des objets à travers la migration et le stockage, et des communications entre espaces d'objets disjoints, bien qu'aucune forme de typage ne soit imposée. Nous présentons les mécanismes de migration, communication, persistence et nommage. Nous tirons aussi les leçons de cette expérience.

Abstract

We present a distributed object-oriented system called SOS. Its goals are to facilitate programming distributed applications, thanks to the "fragmented object" mechanism. SOS provides generic support (invocation, identification, migration, and storage) for application objects. We present the object model of SOS, comprising elementary and fragmented objects. The model maintains the type integrity of objects across migration and storage, and of communication between distinct address spaces, even though SOS imposes no type model. We present the support mechanisms for migration, communication, persistence, and naming. We conclude with an assessment of the lessons learned.

1 Introduction

La méthodologie de programmation par objets se répand de plus en plus. Il existe de nombreux langages à objets, tels que Smalltalk [Goldberg 1983], Eiffel [Meyer 1987], C++ [Stroustrup 1985], CLOS [Gabriel 1989], GUIDE [Decouchant 1989], etc. Chaque langage implémente son propre modèle d'objets. Chacun s'adapte de façon différente au système d'exploitation (SE) existant.

La notion *d'objet* permet de penser différemment la frontière entre système d'exploitation (SE) et application. Tout d'abord, la programmation par objets permet de structurer proprement un SE. Le meilleur exemple est le système Choices [Campbell 1989], structuré en une hiérarchie de types et d'implémentations, correspondant aux notions habituelles de processus, mémoire, etc. Par un choix judicieux d'éléments de cette hiérarchie, on peut configurer des SE différents, adaptés à une configuration matérielle particulière, à un besoin d'une application, ou à une certaine norme.

Ce qui est particulièrement nouveau, c'est que l'application de l'approche objet conduit les SE à offrir des *services de gestion des objets.*

Le thème de recherche du groupe SOR de l'INRIA est la mise en œuvre d'un système d'exploitation réparti à objets (SERO), offrant une gestion des objets commune à toutes les applications et à tous les langages. Un SERO facilite la réalisation de compilateurs pour langages à objets, rend les applications plus efficaces, et permet la communication par partage d'objets.

Un premier système, SOS, visant ces objectifs, a été réalisé. Dans SOS une application est un ensemble d'objets. Le SE, structuré de la même manière, est un ensemble de services de gestion d'objets élémentaires : allocation et destruction, prérequis, appels ascendants, migration, communication protégée. Les applications communiquent en partageant des *objets fragmentés* (OF). Pour constituer réellement un système d'exploitation, ces services doivent être suffisamment complets, de bas niveau, génériques, indépendants d'un langage particulier, et efficaces.

SOS a pour base des "objets élémentaires", modèle à la fois simple et puissant. Un objet élémentaire est une entité typée et structurée, définie de façon quelconque par le programmeur d'application. Un objet élémentaire peut *migrer* entre espaces d'adressage, et peut être *stocké* sur disque. Une granularité raisonnable de ces objets est de l'ordre d'une centaine d'octets et plus. Des objets composés sont construits à partir des mécanismes de

gestion des objets élémentaires.

SOS encourage l'usage du *principe du mandataire* [Shapiro 1986] pour la structuration des applications réparties. Le principe étend le concept d'objet à celui d'objet réparti ou *fragmenté*. Un OF est un objet dont la représentation est composée de *fragments* répartis sur le réseau. La notion d'OF regroupe, dans une même abstraction, les notions d'objet partagé, de capacité, de talon (stub), de réplication, et de partition fonctionnelle.

Extérieurement, un objet fragmenté est un objet comme un autre, mais il est composé de *fragments* (des objets élémentaires) répartis à travers le réseau. Son interface est fournie par ses fragments locaux, appelés mandataires. Entre fragments d'un OF, on peut établir des canaux de communication. Un canal est lui-même un objet, ce qui permet d'assigner une sémantique à la communication. Un OF est typiquement créé par le mécanisme de *migration* qui permet d'essaimer des fragments vers ses clients. La notion d'objet fragmenté garantit que la communication est fortement typée, bien qu'elle puisse emprunter des canaux non typés.

SOS est construit à partir de ses propres mécanismes : tous les services système de SOS sont réalisés par des objets fragmentés avec des mandataires locaux. SOS est écrit en C++, et est prototypé au-dessus du sytème Unix.

Ce papier présente en parallèle dans chacune des sections qui suivent, le système SOS, les choix qui ont présidé à sa conception, et un bilan du prototype.

Le plan de cet article est le suivant. La prochaine section fournit un aperçu des principaux concepts de SOS. La section 3 explique ensuite les objets élémentaires. Elle est suivie par une étude des objets fragmentés en section 4. Puis, la section 5 détaille la migration d'objet. Les sections 6, 7, et 8 décrivent les principaux services de SOS : communication, persistence et nommage. Finalement, la section 9 fait un bilan de l'architecture et de la réalisation du prototype.

2 Vue générale de SOS

SOS est un sytème d'exploitation réparti à objets. L'usager peut définir des objets quelconques, qui seront gérés par SOS. Ses mécanismes permettent la création, la destruction, la migration, le stockage, la localisation, la communication, le nommage d'objets, fragmentés ou non.

2.1 Principaux concepts

Un *objet* est un ensemble de données et d'opérations arbitrairement défini par l'usager. Dans cet article, nous nous intéressons uniquement aux objets gérés par SOS, et que nous appelons des *objets SOS*.[1] Nous distinguons *objets élémentaires* et *objets fragmentés*.

Un objet élémentaire, objet dont la représentation (les données et le code) est localisée dans un seul espace d'adressage, est caractérisé pour SOS par un descripteur.

La partie données d'un objet est accédée uniquement au moyen de son interface procédurale, fortement typée. Un objet accède aux services système en appelant la primitive appropriée : nous appelons cela un *appel descendant*. Réciproquement, le système peut invoquer au moyen d'un *appel ascendant*, un ensemble restreint et bien connu de procédures de l'objet.

Un objet s'exécute dans un *contexte* ou espace d'adressage virtuel. Un contexte peut contenir un nombre quelconque d'objets élémentaires. Les objets élémentaires peuvent migrer d'un contexte à un autre : à un moment donné, un objet élémentaire est actif dans un seul contexte, ou bien stocké sur disque.

Chaque objet possède un identificateur unique, appelé *OID concret*. Un objet est désigné à l'intérieur d'un contexte par son adresse, ou globalement par une *référence*, contenant son OID concret, une indication de localisation, et une partie opaque utilisable par le programmeur.

A partir des objets élémentaires, il est possible de construire des *objets composés*, avec des segments de données multiples connectés entre eux par des pointeurs. Ils seront considérés dans la section 7.

SOS étend la notion d'objet à celui d'objet *réparti* ou *fragmenté*. Un objet fragmenté (OF) est réalisé par un groupe d'objets élémentaires, ses "fragments", éventuellement localisés dans différents contextes, sur différents sites. Sa représentation est l'union de tous les fragments. De la même manière qu'un objet élémentaire peut accéder à sa propre représentation, sans passer par son interface procédurale, les fragments individuels d'un OF sont autorisés à communiquer d'un contexte à l'autre, par des canaux non typés : par messages (*invocation inter-contextes*), par mémoire parta-

[1] Il existe par ailleurs des objets, qui ne migreront pas et ne seront pas stockés de façon indépendante. Ceux-ci n'ont pas à être connus du système; nous les appelons *objets simples*.

gée, etc. Les objets qui ne sont pas des fragments d'un même OF ne peuvent communiquer de cette manière.

Un fragment peut créer et ajouter un nouveau fragment à l'OF, et l'exporter vers un autre contexte. L'appartenance à un OF est préservée par la migration, ce qui permet à un objet fragmenté de s'accroître par essaimage.

Les applications réparties, ainsi que les services systèmes de SOS, sont structurés en objets fragmentés. Trois sortes d'objets élémentaires participent aux OF : les serveurs, les mandataires et les fournisseurs. Un mandataire est un objet représentant le service auprès d'un client [Shapiro 1986]. Chaque client, désirant accéder à un service, doit acquérir un mandataire pour ce service dans son contexte. Le mandataire constitue sa seule interface avec le service. Un mandataire peut traiter les requêtes localement, ou bien les retransmettre au serveur distant (*cf.* figure 1). Un serveur est un objet capable de répondre aux requêtes des mandataires. Un fournisseur est responsable de la fourniture de mandataires à la demande des clients.

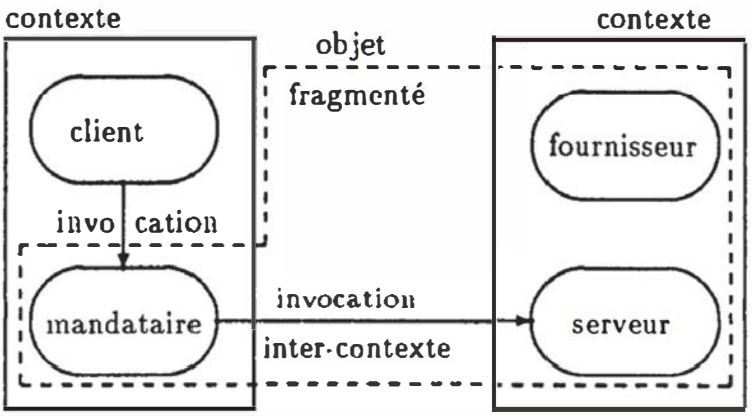

Figure 1 : Le principe du mandataire : un service réalisé par un objet fragmenté.

Les OF et les mandataires constituent un outil puissant et souple pour la structuration des applications réparties.

2.2 Le prototype

Notre prototype de SOS est implémenté en C++ au-dessus du système Unix (SunOS). Cet article décrit sa version 5, disponible au premier trimestre 1990.

SOS comprend un noyau et un ensemble de services système, s'exécutant sur celui-ci. Le noyau fournit des espaces d'adressage séparés (les contextes), des processus légers dans les contextes (les tâches), et l'invocation inter-contextes. La programmation pour SOS est facilitée par l'usage de bibliothèques prédéfinies et d'un compilateur C++ modifié.

Les objets SOS sont des instances de la classe prédéfinie sosObject, ou d'une classe compatible avec celle-ci. C++ dispose de procédures, dites *virtuelles* [Gautron 1987a], liées dynamiquement aux objets, et pour lesquelles une classe dérivée peut redéfinir les actions. Les appels ascendants sont effectués par un appel à une procédure virtuelle de l'objet cible. Ce mécanisme permet au programmeur de définir pour chaque classe d'objet les "procédures ascendantes" adaptées à sa sémantique.

Les espaces d'adressage sont fournis par le système Unix. Les tâches sont réalisées par la bibliothèque des tâches de C++ [Stroustrup 1982] légèrement modifiée. La gestion des contextes est effectuée par un processus Unix appelé sos. L'invocation inter-contextes utilise les "sockets" de type "stream" du domaine Unix.

Sur chaque machine, sos démarre automatiquement les services système. Les applications sont exécutées à partir du shell Unix, ou du débogueur.

Les services systèmes sont structurés sous forme d'Objets Fragmentés.

Quatre services de base sont disponibles :

Le service d'accointances. C'est le gestionnaire d'objets. Il se charge de la localisation et de la migration des objets, en coopération avec les services de communication et de stockage. Chaque contexte dispose à sa création d'un mandataire du service d'accointances. Celui-ci offre au contexte les opérations de base sur les objets élémentaires. (*cf.* sections 3, 4 et 5). Le service d'accointances est le cœur de SOS.

Le service de communication. Le service de communication s'occupe de la communication entre sites et des protocoles d'invocation. Il offre une boîte de protocoles permettant l'appel de procédures distantes, la diffusion, l'adressage fonctionnel, etc. (*cf.* section 6).

Le service de stockage. Le service de stockage est responsable de la persistence des objets. Il définit un ensemble minimal d'outils simples et génériques, afin de rendre quasi-automatique le stockage des objets composés typés (*cf.* section 7).

Le service de nommage. Le service de nommage gère la correspondance entre noms symboliques et références. Tout objet SOS, élémentaire ou fragmenté, peut être nommé. Le service de nommage permet à ses clients de construire leur propre vue de l'espace de nommage (*cf.* section 8).

Après une comparaison avec des travaux similaires, nous détaillerons l'architecture et la mise en œuvre du prototype, que nous évaluerons à la lumière de notre expérience. Pour une introduction plus détaillée à SOS, se reporter à [Shapiro 1989, SOR 1988].

2.3 Comparaison avec des travaux similaires

Emerald (Université de Washington) est un langage à objets pour la programmation d'applications réparties. Il est caractérisé par ses objets mobiles de granularité fine [Jul 1988]. Le compilateur transforme la représentation des objets, définis par l'usager, dans le but de faciliter la migration. Les premiers octets d'un objet sont un descripteur standard, et tous les champs de même type sont groupés ensemble. Conceptuellement, tous les objets coexistent dans un unique espace d'adressage, couvrant tout le réseau. Une

référence à un objet est globale, mais une référence locale est optimisée sous forme de pointeur.

Au contraire d'Emerald, SOS est un système d'exploitation et non un langage. Nous n'imposons aucun format particulier pour la représentation des objets. Les informations système sont bien séparées des données définies par le programmeur, et le système exécute des appels montants sur les objets. Plutôt qu'un unique espace d'adressage, nous offrons un univers *structuré*.

Choices (Université d'Illinois) est une famille de systèmes d'exploitation, conçue et structurée selon l'approche objet, les services proposés aux applications étant assez conventionnels [Campbell 1989]. A l'inverse, dans SOS, nous avons porté l'accent plus sur la réalisation de nouveaux services, facilitant l'implémentation d'applications réparties à objets, que sur l'architecture interne.

Clouds (Institut de Technologie de Georgie) est un autre système d'exploitation réparti à objets. Son but principal est l'intégration de la sûreté de fonctionnement dans les couches basses du système [McKendry 1985]. A l'inverse, il n'y a pas actuellement de mécanismes pour la sûreté de fonctionnement dans SOS. On peut supposer que leurs objets sont généralement plus gros que les nôtres, puisqu'un "objet Clouds" s'exécute dans son propre espace d'adressage.

Guide/Commandos (LGI, Grenoble) est un environnement de programmation réparti, conçu à partir d'un langage, et destiné à des applications bureautiques [Decouchant 1989]. L'univers est structuré en espaces d'adressage multi-machines appelés "domaines". Quand un domaine doit accéder à un objet situé sur une autre machine, il s'étend sur cette machine, liant cet objet. Cette structure est plus simple d'utilisation que les Objets Fragmentés de SOS : toutefois la structuration en OF se prête mieux aux systèmes de grande échelle et à la réplication.

Gothic (IRISA Rennes) est un langage et un système pour programmes répartis sûrs de fonctionnement [Banâtre 1986]. Il est basé sur une théorie des objets fragmentés, invoqués par "multi-fonctions" (invocations sans effets de bord, coordonnées sur un ensemble de sites), reconnues par le langage. Nos objets fragmentés sont "ad-hoc" mais plus souples. Nous empruntons le terme d'objet fragmenté à Gothic.

3 Les Objets Elémentaires

3.1 Présentation des Objets Elémentaires

L'entité de base gérée par le gérant d'objets, dit "Service d'Accointances" ou SA, est l'*Objet Elémentaire* (OE). Nous avons rendu l'Objet Elémentaire aussi simple que possible, tel un "plus grand dénominateur commun" pour tous les usages.

Un OE est un segment de données unique, défini par l'utilisateur, auquel est associé un descripteur système.

A un moment donné, un Objet Elémentaire existe dans un contexte unique, sur un seul site. Chaque OE est différent de tout autre et est caractérisé par un identificateur unique appelé *OID concret* (où OID veut dire "identifiant d'objet"). SOS connaît un Objet Elémentaire par le biais de son descripteur d'accointance, ou DA. Il existe une table des DA par contexte, gérée par le mandataire du service d'accointances dans ce contexte.

Le DA d'un objet contient les informations suivantes:

- son OID concret et (éventuellement) une liste d'*OID de groupe* (*cf.* section 4.2),
- la référence de son *objet code* (*cf.* section 3.3) et (éventuellement) la liste de ses *prérequis* (*cf.* section 5.3),
- l'adresse et la taille de son segment direct (*cf.* section 7.2) ,
- (éventuellement) sa liste de *canaux* (*cf.* section 4.2) .

Les termes en italiques seront détaillés par la suite.

3.2 Création d'Objets Elémentaires

Un objet SOS est instance de la classe sosObject, ou d'une classe qui en dérive. Un objet est donc à sa création, soit un objet simple, soit un objet SOS, et le reste toute la durée de son existence. Nous constatons que ce schéma est un peu rigide; il serait utile de permettre à un objet simple de se faire connaître du système et accéder dynamiquement au statut d'objet SOS.

En C++, la création d'un objet est assurée par une procédure spéciale de sa classe, appelée *constructeur*. L'instanciation d'une classe dérivant d'une autre classe, entraîne des appels aux constructeurs de ces classes dans un ordre bien déterminé. Tout objet SOS étant instance d'une classe dérivant de **sosObject**, le constructeur de la classe mère **sosObject** est donc appelé implicitement à sa création. Ce constructeur alloue un DA libre, et le remplit avec un OID nouvellement alloué, ainsi qu'avec l'adresse et la taille du segment direct.

Ce schéma de création d'objet est appréciable pour le programmeur C++, car le compilateur appelle automatiquement les primitives de création et de destruction des objets.

Un inconvénient est que l'interface du système n'est pas clairement identifiée, et n'est pas tout-à-fait indépendante du langage. Par exemple, l'allocation de mémoire est incluse dans le constructeur. Par contre, la référence du code[2] ne peut pas être établie par le constructeur ; un appel séparé à une procédure de déclaration d'objet code associé est nécessaire avant toute migration.

3.3 Les objets code

A tout objet élémentaire est attaché un autre OE appelé *objet code* (par appel d'une primitive spéciale). Comme nous le verrons au paragraphe 5.3, cet attachement garantit que le type de l'objet est préservé à travers la migration et le stockage, bien que SOS n'ait pas de mécanisme de type explicite.

Un *objet code* est lui-même un OE, instance de la classe prédéfinie **code**. Il contient le code compilé des opérations d'une certaine classe. Par exemple, le code d'une instance d'une classe **X**, définie par l'usager, serait géré par une instance de la classe code **code_pour_X**. Pour la migration, le DA d'un objet doit contenir une référence vers l'objet code correspondant.

Le système traite une instance de la classe **code** de façon identique à tout autre objet. C'est un exemple de la puissance de notre modèle d'Objet Elémentaire. Comme nous le verrons dans la section 5.3, l'édition de liens dynamique et la vérification de types sont automatiques; elles ne sont pas vissées dans le système, mais sont effectuées par le réinitialiseur (*cf.* section 5.2) de la classe **code**. Ceci est un exemple de l'uniformité de traitement

[2]Sans laquelle l'objet ne peut être migré (*cf.* section 3.3).

et de réutilisation de fonctionnalités génériques, au service d'une sémantique spécifique.

4 Les Objets Fragmentés

4.1 Présentation des Objets Fragmentés

Les applications réparties sont structurées en *Objets Fragmentés (OF)*. Un OF permet la communication et la coopération réparties, sans interférence étrangère. Un OF est réalisé par un groupe d'Objets Elémentaires, appelés ses fragments, ayant le privilège de communiquer ensemble. On peut aussi voir un OF comme un objet unique dont la représentation est fragmentée. Un client de l'OF y accède localement, par l'interface procédurale fortement typée, fournie par son fragment local de l'OF, dit *mandataire*. L'interface publique de l'OF est une sorte d'union des interfaces de ses fragments.

De la mêmemanièrequ'un Objet Elémentaire peut accéder directement à sa propre représentation, sans passer par son interface publique, un fragment peut accéder à la représentation interne du groupe. Ainsi, les fragments peuvent communiquer, soit en utilisant la mémoire partagée, soit par envoi de messages, soit par fichier partagé, etc. Par contre, deux OE qui ne sont pas membres d'un même OF ne peuvent communiquer que par leur interface procédurale, et seulement s'ils sont situés dans le même contexte. Ainsi, la communication inter-contextes est fortement typée, bien qu'elle emprunte des canaux non typés, et sans que SOS n'ait à proprement parler de notion de type.

4.2 Gestion d'Objet Fragmenté

Le but d'un Objet Fragmenté est de fournir des facilités de communication et de coopération aux applications réparties.

Conceptuellement, un OF met en œuvre un domaine de protection, dans lequel on entre par l'invocation d'un des mandataires locaux.

Un OF est caractérisé par le fait que chaque fragment possède un *OID de groupe*, en plus de son OID concret. Les OE ayant un OID de groupe en commun, appartiennent au même OF. Un OE peut éventuellement appartenir à plusieurs OF.

Un OE peut réaliser une invocation inter-contextes ("Remote Procedure Call" sur une même machine) vers un autre OE membre du même OF, par opération sur un *canal*. Un canal est un objet (simple), matérialisé par un champ du DA de l'objet source et pointant sur le DA destinataire. Les canaux vers des sites distants, ainsi que les canaux réalisant des protocoles particuliers, sont expliqués dans la section 6.

4.3 Identification des objets

Un objet s'identifie de deux manières différentes : on peut le désigner par son adresse (ou, pour un OF, par l'adresse de son mandataire), ou par une *référence* indépendante de la localisation de l'objet (contenant un de ses OID). Une adresse n'a aucun sens en dehors de son contexte. Elle doit être explicitement traduite en une référence, par exemple si elle doit être passée dans un message. L'identification des objets dans SOS n'est donc pas uniforme.

Le client d'un service n'a pas besoin de connaître ces deux identités, puisqu'il accède au service uniquement en utilisant l'adresse du mandataire local.

Au contraire, le programmeur d'un Objet Fragmenté doit les maîtriser. A l'intérieur de l'Objet Fragmenté, certains Objets Elémentaires sont locaux les uns aux autres, et d'autres se trouvent dans des contextes séparés. La communication entre les premiers utilise des adresses et des invocations locales, alors qu'entre les seconds, elle utilise les références et l'invocation inter-contextes sur un canal.

SOS offre des outils pour pallier à cet inconvénient. Des *pointeurs permanents* (section 7) effectuent automatiquement la conversion entre les références et les adresses. Les objets canal aident à dissimuler la distinction entre les différents modes d'invocation. Finalement, les dépendances (section 6.5) remplacent l'invocation explicite par un mécanisme plus uniforme de propagation de changements d'état.

4.4 Générateur de fragments

Jusqu'a récemment, chaque type de fragment devait être programmé manuellement. Le programmeur devait lui-même assurer la cohérence des interfaces et des données, même à l'intérieur d'un type d'objet fragmenté donné. De même que de nombreux systèmes ont des générateurs de talons (stub

generators [Jones 1985]), nous proposons maintenant FOG, un langage et un compilateur pour faciliter l'écriture d'applications réparties structurées en OF [Gourhant 1990]. A partir d'une description déclarative du type d'OF et de sa structure, il génère automatiquement du code pour les divers types d'OE le composant, assurant leur cohérence de type. Il prend en charge les aspects communs de programmation des fragments, c'est-à-dire l'allocation d'OID de groupe, la mise en place des canaux, et l'empaquetage/dépaquetage des messages. Finalement, le générateur aide à coordonner les changements d'état entre les fragments.

4.5 Protection des objets fragmentés

Seul l'Objet Elémentaire courant doit avoir accès à ses propres canaux. Pour toute invocation, le compilateur C++ ajoute l'adresse de l'objet invoqué comme premier argument (invisible à l'utilisateur). Ainsi, sur occurrence d'une invocation, le noyau vérifie que l'identité de l'appelant dans les données du canal, est égale à l'argument "objet courant" de l'avant-dernière entrée de la pile d'exécution. Ceci constitue un faible mécanisme de protection des domaines des OF, mais notre environnement (Unix sur matériel standard), ainsi que la granularité des objets, ne nous permettaient pas la réalisation d'un mécanisme de protection à l'exécution plus sûr, à un coût raisonnable.

Dans le but de fournir une certaine protection aux objets fragmentés, l'ajout d'un nouveau fragment à un OF existant, ainsi que la création des canaux, ne peut se faire qu'entre objets instanciés dans le même contexte. La façon normale de créer un OF est de créer des mandataires localement, puis de les faire migrer (*cf.* section 5) dans d'autres contextes, l'appartenance à un OF ainsi que les canaux étant préservés au cours de la migration.

Cependant, le fait que l'OF n'est pas vraiment protégé lors de l'exécution, ainsi que l'identification non uniforme, sont deux symptômes d'une organisation inadéquate de la mémoire.

Une organisation de mémoire structurée, comme celle offerte par les machines à capacités, pourrait améliorer la protection à l'exécution. Une idée plus attrayante est de garantir l'intégrité du groupe dès la compilation. Le langage FOG est une étape dans cette direction.

5 Migration des Objets Elémentaires

Tout service réparti est réalisé par un Objet Fragmenté. Son interface publique sera fournie localement par ses fragments, agissant en tant que mandataires pour le service. Pour accéder à un service, un client doit acquérir un mandataire approprié. Cela est fait de façon dynamique par *migration* d'un fragment dans le contexte du client.

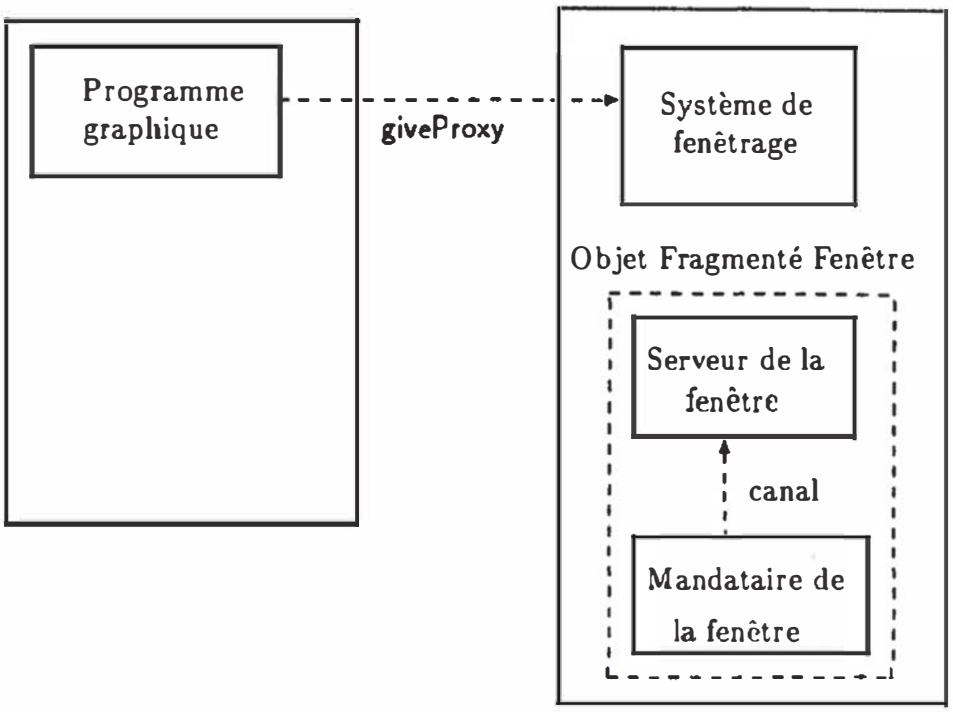

Figure 2 : Avant migration : le gestionnaire de fenêtrage a créé un mandataire et un serveur de la fenêtre, et les a connectés ensemble.

5.1 Exemple de migration

Considérons l'exemple d'une demande d'ouverture de fenêtre sur l'écran. Dans SOS, ce sera une demande d'importation d'un mandataire de fenêtre, adressée au gestionnaire de fenêtrage, fournisseur de mandataires. Pour ouvrir la fenêtre, ce dernier va créer un mandataire de fenêtre M (qui sera

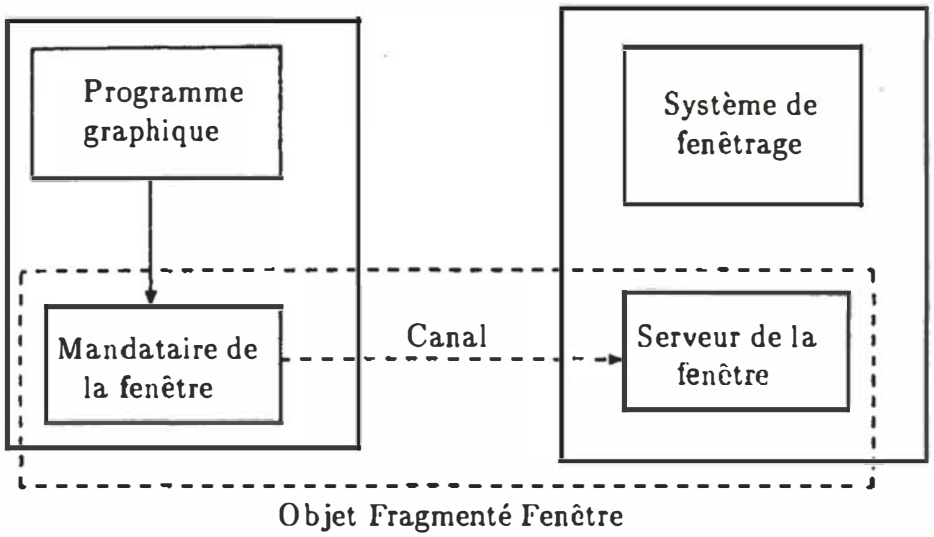

Figure 3 : Après migration : le mandataire de la fenêtre est dans le contexte du client et reste connecté au serveur de la fenêtre

exporté vers le demandeur, comme interface de sa fenêtre), et un serveur de la fenêtre S, qui exécutera les commandes graphiques. Cet exemple est illustré par les figures 2 et 3.

5.2 Les mécanismes de la migration

La migration des objets dans SOS repose sur deux types de mécanismes. D'une part, un mécanisme transport, et d'autre part, un mécanisme de réinitialisation de l'objet transporté. Ces mécanismes sont détaillés ci-dessous. Il existe deux variantes de la migration: par déplacement ou par copie.

Transport (importation ou exportation)

L'importation d'un objet mandataire est à l'initiative d'un client : désirant accéder à un service, il requiert l'importation d'un mandataire de ce service dans son contexte. Il émet une demande d'importation en exécutant la procédure import (référence, arguments) de son mandataire local du SA. Celui-ci transmet la requête (par invocation inter-contextes) au mandataire du SA du contexte fournisseur. Ce dernier mandataire fera un appel ascendant à la procédure **giveProxy (client, arguments)** du fournisseur. Cette

procédure utilisateur se charge de créer ou sélectionner un objet candidat à la migration. Elle lui attachera son objet code et éventuellement des objets prérequis (voir section 5.3), et établira un canal entre cet objet et le serveur. Ensuite, l'importation de l'objet est prise en charge par le Service d'Accointances par copie ou déplacement.

Outre l'importation, seul mécanisme envisagé au départ, les besoins de certaines applications ont nécessité un mécanisme *d'exportation*, où l'initiative de la migration revient au fournisseur. Un Objet Fragmenté a la possibilité d'exporter un objet vers un de ses fragments avec lequel il est relié par un canal. Une exportation ressemble beaucoup à une invocation sur ce canal ; le destinataire reçoit, par appel ascendant à sa procédure stub, un message d'invocation particulier lui signalant l'arrivée d'un objet exporté. La migration (par copie ou déplacement) est, ici encore, prise en charge par le SA.

L'algorithme de migration d'un objet comporte plusieurs étapes, à savoir : la copie ou le déplacement de ses données et de son DA dans le contexte destinataire, l'importation de ses prérequis si nécessaire (*cf.* section 5.3), et pour terminer l'appel de sa *procédure de réinitialisation*.

La réinitialisation

La réinitialisation est l'étape finale du processus de migration d'un objet. Une fois ce dernier installé dans le contexte destinataire, sa procédure de réinitialisation (un constructeur de la classe correspondante) est appelée de façon ascendante. Cette procédure se charge par exemple de mettre à jour les pointeurs, ou bien de requérir des importations supplémentaires.

5.3 Migration du code et des prérequis

Les prérequis sont des objets dont l'objet migré a besoin dans son environnement pour fonctionner. Par exemple, le code d'un objet est un type particulier de prérequis. A l'importation d'un objet, ses prérequis sont importés récursivement avant d'appeler le réinitialiseur. Un prérequis n'est importé que s'il n'est pas déjà présent, même sous forme de mandataire ; cela permet par exemple à deux objets importés de partager le même code, s'ils sont d'implémentation identique. De même, le code des mandataires peut être lié statiquement, sans perte de fonctionnalité.

Puisqu'un prérequis est importé selon le même algorithme que les autres objets, sa procédure de réinitialisation est appelée à la fin de sa propre migra-

tion. Le réinitialiseur pour un objet code est une *édition de liens dynamique et une vérification de types* [Gautron 1987b]. C'est grâce à ce mécanisme que le typage fort des objets, imposé par le compilateur C++, est préservé au cours de la migration, même entre processus compilés séparément. Et ceci, bien que SOS n'impose aucun schéma ou mécanisme de typage particulier.

L'algorithme utilisé est spécifique au langage C++, mais d'autres langages pourraient être intégrés en dérivant une réalisation différente de la classe code.

5.4 Discussion

La migration est complètement générique, au moyen des appels ascendants. Ainsi, un premier appel ascendant à givePoxy initialise une importation; un second appel ascendant vers le réinitialiseur termine la migration. Le mécanisme des objets code et des objets prérequis contribuent aussi à ce caractère générique. La primitive givePoxy et le réinitialiseur sont des procédures programmables par l'utilisateur, et ce pour tout type d'objets.

Cette possibilité d'extension d'un mécanisme système avec une *sémantique définie par le programmeur* est un concept très important. Des objets quelconques peuvent être migrés : la sémantique de leur migration est spécifique à leur type, bien que basée sur un mécanisme système unique.

La force de ce modèle est que la migration préserve la sémantique et le type des objets, définis par l'utilisateur. Les prérequis sont des Objets Elémentaires comme les autres. L'édition de liens dynamique et la vérification de types sont automatiques, sans être câblées.

6 La communication

Le seul protocole offert par le noyau est une invocation inter-contextes, le long d'un canal à l'intérieur d'une même machine. Les accès distants (inter-sites) et les autres protocoles (tels que les envois multiples) sont réalisés par des *objets protocole*, réalisés par le Service de Communication.

6.1 Boîte à outils de protocoles

Le Service de Communication [Makpangou 1988, Makpangou 1989] est une boîte à outils de protocoles.

Dans l'état actuel, cette boîte à outils offre la communication simple (1 à 1) et la diffusion de groupe (1 à N). L'un ou l'autre peuvent être synchrones ou asynchrones. Les résultats des communications asynchrones et multiples sont gérés par un *objet invocation*.

La communication 1 à N est basée sur le concept de *famille*. Une famille est un sous-ensemble d'un Objet Fragmenté. Ses *membres* communiquent par diffusion. Une famille comprend des opérations pour ajouter et retirer des membres, et une interface d'invocation multiple pour ses membres.

Pour chaque type de protocole d'invocation (par exemple appel de procédure à distance [Nelson 1981], ou diffusion, synchrone), il existe une classe correspondante, qui peut être instanciée dans le contexte du Service de Communication. Ces classes de base de la boîte à outils peuvent être étendues (par héritage ou redéfinition) ou composées, pour construire de nouveaux types de protocoles.

6.2 Interface applicative

L'interface d'application au Service de Communication contient des *objets canaux*, pour communiquer entre les fragments. Un objet canal encapsule l'accès aux objets protocole. Pour utiliser un protocole, une application instancie un objet canal entre les objets d'application concernés. Le canal est un mandataire de l'objet protocole désiré. Ce dernier est instancié dans le contexte du Service de Communication de la machine locale.

Deux types compatibles d'objets canaux sont couramment définis, interfaçant les protocoles 1 à 1 et 1 à N. Une invocation 1 à 1 sur un canal 1 à N aura pour effet de sélectionner arbitrairement un membre de la famille comme récepteur du message. Ceci est appelé de *l'adressage fonctionnel* [Legatheaux Martins 1987]. En outre, un membre particulier peut être invoqué en spécifiant son OID concret dans le message d'invocation. Une invocation de type 1 à N réalise l'invocation de tous les membres de la famille, ce qui constitue de *l'adressage de groupe*.

6.3 Structure interne

Un objet protocole constitue une couche en-dessous de l'application qu'il sert ; ceci est illustré par la figure 4. Un objet protocole est un Objet fragmenté composé d'objets protocole élémentaires coopérants, instanciés dans

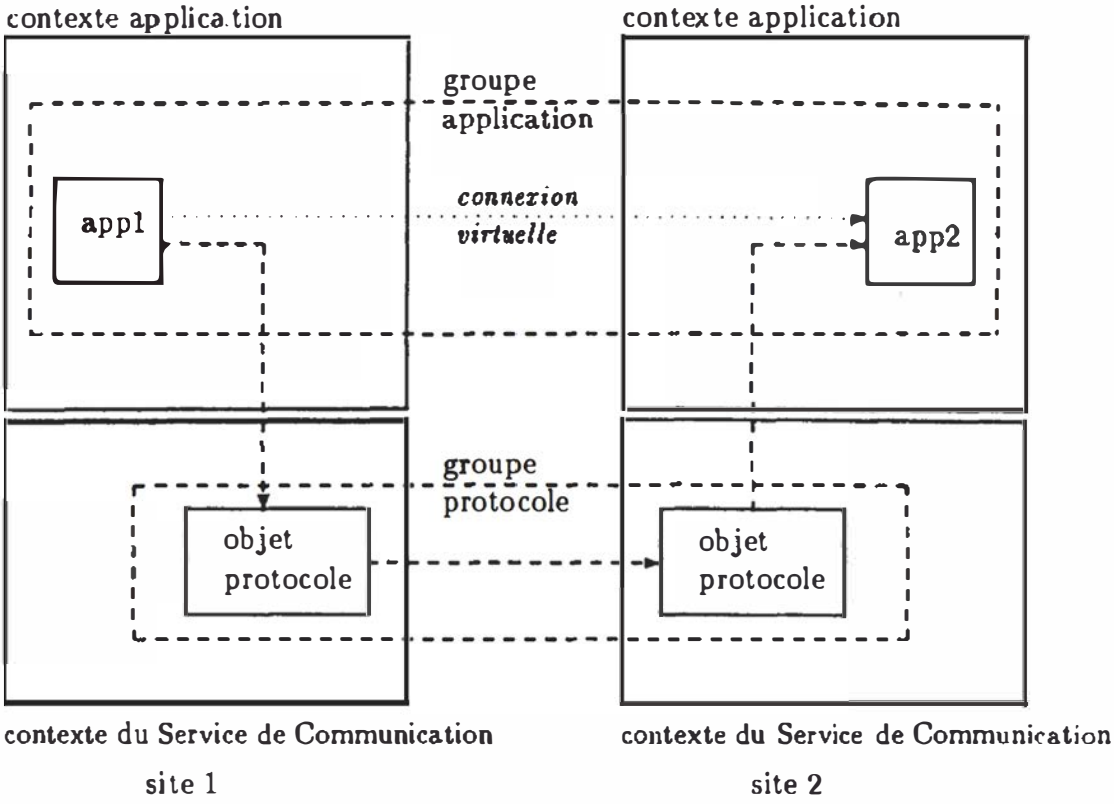

Figure 4 : Un objet protocole réparti, en-dessous des fragments d'une application.

le contexte du Service de Communication de chacun des sites.

Un objet protocole a le privilège de pouvoir être source ou cible d'un objet canal, même s'il n'est pas membre du groupe de l'application qui l'utilise. Autrement, un objet protocole est un Objet Fragmenté standard.

6.4 Bilan

A présent, il n'y a pas assez de protocoles dans la boîte à outils. Toutefois, son extension est facile, car les communications sont structurées en plusieurs niveaux : les Objets Fragmentés gèrent les domaines de protection; les *objets canaux* interfacent les protocoles d'invocation; les *objets invocation* sont utilisés pour obtenir des réponses multiples et gérer les envois asynchrones.

Au-dessus de ces niveaux, le "générateur de fragments" que nous proposons à la section 4.4 gère le niveau "méthodes", en générant automatiquement les procédures d'interface d'un Objet Fragmenté.

Finalement, la conception hiérarchique du Service de Communication facilite la réutilisation. Les invocations multiples réutilisent le code implémenté pour les invocations simples, et sont distinctes du concept de famille. Nous comptons réutiliser ces objets pour implémenter de nouveaux protocoles, comme ceux d'Isis [Joseph 1988].

6.5 Les dépendances

Jusqu'ici, nous avons décrit les mécanismes pour structurer et gérer les communications entre objets apparentés. Du point de vue du système, aucune sémantique n'a besoin d'être associée aux invocations entre objets; c'est une partie du protocole interne de l'Objet Fragmenté. C'est un mécanisme simple et flexible, sur lequel de nombreuses applications peuvent être construites. Cependant, il y a des situations (ou des événements) anormaux qui ne peuvent être gérés de cette façon.

Par exemple, supposons que nous ayons un serveur de fichiers, qui contrôle le partage en utilisant des verrous. Sur demande de lecture ou d'écriture d'un fichier, un mandataire est exporté dans le contexte client, détenant un verrou approprié. Si une panne se produit du côté du client, le serveur de fichiers doit en être informé afin de détruire les informations résiduelles, et de rendre le fichier à nouveau utilisable pour les demandes suivantes. Dans ce but, nous définissons le mécanisme de *dépendance*, qui permet de:

1. déclarer des objets comme parties d'une *famille de dépendance* commune,
2. détecter et prendre en compte des situations qui relèvent d'une famille de dépendance donnée, et
3. propager les événements correspondants à l'intérieur d'une famille de dépendance.

Deux types d'événements sont reconnus :

- les événements détectables par le système, correspondant à une terminaison *normale* ou *anormale*, et

- les événements définis par l'utilisateur.

Bien que les deux types d'événements soient traités de façon uniforme, les motivations pour chacun des types sont différentes. L'événement de terminaison anormale est généré sur panne de site ou de contexte : seul le système peut détecter ce genre d'événements. La terminaison normale pourrait être gérée manuellement ; mais par commodité, nous permettons d'utiliser le même mécanisme. Enfin, les événements utilisateur facilitent la gestion des signaux logiciels. Une primitive permet de déclarer un changement de l'état interne d'un objet, à propager selon le mécanisme de dépendance.

7 La persistence

7.1 Présentation

Le Service de Stockage gère, de manière générique, le stockage physique sur disque d'objets *typés* et *composites*. Une fois stockés, les objets deviennent *permanents*. Leur représentation stockée ne peut jamais être détruite.

Notre but premier était de réaliser un stockage à un seul niveau, intégrant de façon transparente la "migration verticale" (depuis et vers le disque) avec la "migration horizontale" (de contexte à contexte).

L'importation verticale, depuis le stockage vers un contexte, est donc identique à l'importation horizontale. Ainsi, par les mêmes mécanismes de prérequis que l'importation (*cf.* section 5.3), le stockage préserve le type et la sémantique des objets stockés. Par contre, pour des raisons historiques, une primitive spéciale est utilisée pour envoyer un objet de la mémoire vers le disque. Cette primitive est optimisée pour stocker seulement les portions modifiées de l'objet.

Pour être permanent, un objet doit hériter de l'interface définie par le type *Objet Permanent*.

Ceci constitue une limitation gênante puisque la permanence est un attribut de compilation, s'appliquant à tous les objets d'une même classe, et non un attribut dynamique. Un mandataire du Service de Stockage est associé à tout objet permanent. Un accès privilégié entre ces deux objets permet d'encapsuler les communications avec le Service de Stockage et de cacher le mandataire de stockage à l'application. Le mandataire est importé de façon

transparente lors de la création, ou lors de l'activation à partir du disque, de l'Objet Permanent.

7.2 Objets composites

Un Objet Permanent peut être composé de plusieurs segments et est alors un *objet composite*. Pour être pris en compte par le Service de Stockage, ces segments doivent être référencés par des "*pointeurs permanents*" spéciaux.

Les pointeurs permanents permettent de construire des objets dont les segments forment un graphe arbitrairement complexe, les segments pointant les uns vers les autres. Pendant la migration verticale d'un Objet Permanent, la structure de son graphe est préservée. Un pointeur permanent est relogeable. Il garde une identification indépendante du contexte, mais comprenant le pointeur réel dans le contexte courant.

Au premier accès par l'objet, le segment référencé est automatiquement extrait du stockage. Inversement, les segments modifiés sont sauvegardés lors d'un point de sauvegarde ("checkpoint"). Les changements dans les données d'un segment ne peuvent pas être automatiquement détectés, car nous ne contrôlons pas la gestion de la mémoire virtuelle d'Unix. C'est donc le programmeur qui doit indiquer au système qu'un segment a été modifié, en marquant le pointeur permanent qui le référence.

7.3 Bilan

La migration verticale n'est pas encore bien intégrée avec la migration horizontale. Un Objet Permanent complexe ne peut être migré horizontalement en entier, que par l'intermédiaire d'un stockage sur disque : l'AS ignore l'existence des pointeurs permanents, et leur gestion est laissée au programmeur. Donc, un défaut du prototype courant est d'être excessivement modulaire: le Service de Stockage (SS) se présente comme un service indépendant au sommet de la gestion de base des objets, et non intégré avec celle-ci. Il n'existe pas de concept commun des segments de mémoire entre le noyau, l'AS, et le SS. Une solution efficace et élégante requerrait le contrôle de la gestion de la mémoire virtuelle, comme cela est fait dans le sous-système COOL [Habert 1989, Habert 1990].

La structure de l'objet devrait être accessible à tous ces services, mais elle se trouve dans la partie de l'utilisateur de l'objet, et n'est accessible au

Service de Stockage que par le biais d'un objet de stockage dédié.

Dans l'avenir, les migrations horizontale et verticale des objets complexes doivent être unifiées. Pour assurer un modèle d'objet plus uniforme, nous devrions unifier les Objets Permanents avec les Objets Elémentaires. La persistence est un mécanisme qui devrait être orthogonal au typage.

8 Le nommage

Le nommage dans SOS comprend une couche de noms internes, les *références* (*cf.* section 4.3), et une seconde couche, les *noms symboliques*. La gestion des noms symboliques est entièrement assurée par le Service de Nommage de SOS [Le Narzul 1989]. Ce service de haut niveau permet aux utilisateurs d'avoir accès à un espace de désignation structuré (contrairement à l'espace des noms internes qui est plat). Sa principale fonction est d'associer des noms symboliques à des références.

8.1 Vues de nommage

Dans un système réparti à objets comme SOS où tout objet peut être désigné par un nom symbolique (site, fichier, processus, etc.), la vision qu'ont les usagers du système est reflétée par l'espace de désignation symbolique.

C'est pour cela que, contrairement aux systèmes traditionnels où tous les usagers partagent le même espace de désignation, nous avons choisi d'associer une "vue de nommage" à chaque client. Cela permet à chacun d'organiser différemment sa vue de nommage en fonction de ses besoins et ainsi de personnaliser son environnement de travail.

8.2 Objets de nommage

Un *objet de nommage* est défini comme étant une entité générale associant des noms symboliques à des références internes. Le plus simple de ces objets est le *catalogue* qui répertorie des objets élémentaires définis par les programmeurs et également d'autres catalogues afin de constituer un arbre de nommage.

Un autre type d'objet de nommage est la *table de montage* qui associe des préfixes de noms symboliques à des objets de nommage. Elle matérialise la vue de nommage d'un usager du système.

L'interprétation d'un nom par une table de montage débute par la recherche du plus long préfixe lui correspondant; la partie du nom non analysée est ensuite interprétée par l'objet de nommage associé au préfixe trouvé.

Un *objet d'union* réalise l'union d'objets de nommage. Ces objets d'union sont inspirés par le mécanisme similaire de Plan 9 [Presotto 1988] et de QuickSilver [Cabrera 1987]. Grâce à ce nouveau type d'objet, il est possible d'unifier le nommage d'objets qui étaient séparés, par exemple pour des raisons administratives. Ainsi, on peut réunir sous un même préfixe des objets processus répertoriés dans des catalogues différents; on obtient ainsi une liste de processus s'exécutant dans le système indépendamment de leur localisation.

Les éventuels conflits de noms sont résolus par l'attribution de priorités aux différents objets de nommage réunis.

8.3 Utilisation du mécanisme du mandataire

La réalisation de cette architecture est facilitée par le mécanisme du mandataire. Chaque client du service de nommage dispose d'un objet mandataire qui décrit sa vue de nommage en encapsulant une table de montage. Ceci est réalisé indépendamment du site et du contexte d'exécution du client, et indépendamment des autres clients. L'accès à un objet de nommage distant, par exemple un catalogue, se fait également par le biais d'un mandataire (importé dans l'espace d'adressage du client) qui redirige les requêtes vers le contexte du catalogue.

9 Conclusion

Nous avons conçu un système d'exploitation à objets réparti, SOS, dont nous avons réalisé un prototype. SOS est conçu de manière à encourager la structuration des applications réparties en Objets Fragmentés (OF). SOS est lui-même constitué d'un ensemble d'OF prédéfinis. Cet article a présenté l'architecture du système SOS ainsi que ses choix de conception. Nous récapitulons ici brièvement les leçons importantes que l'on peut tirer de cette expérience.

Un système à objets diffère des systèmes traditionnels par l'approche objet utilisée pour sa conception et réalisation. Ainsi, dans SOS, le système de

communication est constitué d'objets protocoles, instances d'une hiérarchie de types de protocoles.

Par ailleurs, dans un systèmes à objets, les utilisateurs ont la possibilité de suppléer les mécanismes système, par des sémantiques ou politiques de leur choix, spécifiques aux différentes catégories d'objets. Dans SOS, ceci est réalisé par le biais d'*appels ascendants* du système vers les objets de l'application, requérant le lancement d'actions spécifiques avant et après la migration, ainsi que sur réception d'une invocation.

Le modèle d'objets de SOS est simple et général. De plus il maintient l'intégrité de type des objets et des communications. Ceci est possible grâce au mécanisme des *prérequis*, qui n'impose aucun mécanisme ou schéma de typage particulier.

Nous concluons de notre expérience qu'un support système pour des objets arbitrairement définis par l'utilisateur est viable et utile. SOS met en œuvre des mécanismes génériques pour la gestion des objets, comme l'identification, la localisation, l'invocation, la migration, le stockage, le nommage et la communication. Le surcoût système fait qu'un objet SOS ne devrait raisonnablement pas être plus petits qu'une centaine d'octets; une application ou un langage génère souvent des objets plus petits, qui seront alors rassemblés dans un seul objet SOS.

Les machines actuelles fonctionnent avec des adresses codées sur 32 bits. Ce nombre est trop petit pour pouvoir réaliser un système dans lequel les objets seraient identifiés uniquement par leur adresse, dans un espace d'adressage unique. Ceci, plus l'existence de deux niveaux de granularité des objets (les objets système et les objets langage) entraîne une identification non uniforme. Tout objet est localement identifié par son adresse; mais les objets système possèdent en outre une identification unique. Il n'existe pas de correspondance évidente entre ces deux identifications.

Dans le but de cacher cette non-uniformité, une technique classique consiste à générer des talons ("stubs"), qui font que les invocations distantes apparaissent comme locales. Dans SOS, nous avons un concept plus général, celui d'*objet fragmenté*, qui est localement représenté par un fragment mandataire. Tout accès à un service, qu'il soit local ou réparti, se fait toujours par invocation d'un objet local. L'existence de caches locaux, de mécanisme de réplication, ou de protocoles spécifiques, entrent naturellement dans le cadre des mandataires; la transparence du réseau est effective mais n'est pas câblée.

Les Objets Fragmentés sont importants pour la structuration des applications réparties. La programmation d'un OF est cependant complexe, et il reste des problèmes de protection des fragments. Le générateur de fragments FOG est une première étape vers une solution aux problèmes soulevés. Une autre possibilité consisterait à réaliser un OF comme un domaine de protection réparti.

Dans un système à objets, les composants communiquent par partage d'objets. Dans SOS, un objet réparti partagé est naturellement réalisé par un OF, en utilisant les fonctionnalités de base offertes par le système. Une de ces fonctionnalités utile est la diffusion atomique, qui garantit que tous les fragments d'un OF ont une vue cohérente de son état. Un autre outil utile serait la mémoire partagée, locale et répartie. Le développement d'autres outils de base pour la réalisation d'applications réparties fait partie de nos futurs travaux.

Pour terminer, la structure en objets de SOS permet d'étendre le système aisément. De cette manière, chaque application paie uniquement le prix de ses besoins spécifiques. Le système a été construit selon ses propres mécanismes. Ceci démontre que les abstractions utilisées sont adaptées à la construction d'un système réparti.

SOS est disponible sous forme source. Cependant quelques fichiers sont protégés par une licence ATT.

Remerciements

Nous tenons à remercier les experts Esprit, MM. Coulouris, Prini et Schindler, ainsi que nos partenaires du projet SOMIW pour leur soutien. Outre les signataires, Vadim Abrossimov a contribué à la conception et la mise en œuvre initiale, Philippe Gautron a réalisé les mécanismes de liaison dynamique et les modifications du compilateur C++, Mesaac Makpangou a conçu et réalisé le Service de Communication, et Andreas Spiracopoulos a fait fonctionner les dépendances.

Table des matières

Références

[Banâtre 1986] Banâtre (Jean-Pierre), Banâtre (Michel) et Ployette (Florimond). – *An overview of the Gothic distributed operating system.* – Rapport de recherche n° 504, INRIA, mars 1986.

[Cabrera 1987] Cabrera (Luis Felipe) et Wyllie (Jim). – *QuickSilver Distributed File Services: An Architecture for Horizontal Growth.* – Research Report n° RJ 5578 (56697), San Jose, CA (USA), IBM Almaden Research Center, avril 1987.

[Campbell 1989] Campbell (Roy H.), Johnston (Gary M.), Madany (Peter W.) et Russo (Vincent F.). – *Principles of Object-Oriented Operating System Design.* – Rapport technique n° R-89-1510, Urbana, Illinois (USA), Department of Computer Science, University of Illinois, avril 1989.

[Decouchant 1989] Decouchant (D.), Duda (A.), Freyssinet (A.), Van (H. Nguyen), Riveill (M.) et de Pina (X. Rousset). – Guide : Un système réparti à objet. *In : Actes Convention Unix 89.* AFUU, pp. 297–316. – Paris, mars 1989.

[Gabriel 1989] Gabriel (Richard P.). – The Common Lisp Object System. *AI Expert*, mars 1989, pp. 54-65.

[Gautron 1987a] Gautron (Philippe) et Habert (Sabine). – Une introduction à C++. *Minis et Micros*, vol. 281 & 282, juin 1987, pp. 40–63.

[Gautron 1987b] Gautron (Philippe) et Shapiro (Marc). – A dynamic link editor for C++. *In : Un recueil de papiers sur le système d'exploitation réparti à objets SOS, Rapport Technique INRIA no. 84.* – Rocquencourt (France), Institut National de la Recherche en Informatique et Automatique, mai 1987.

[Goldberg 1983] Goldberg (Adele) et Robson (David). – *Smalltalk-80: The Language and its Implementation.* – Addison-Wesley, 1983.

[Gourhant 1990] Gourhant (Yvon) et Shapiro (Marc). – FOG/C++: a fragmented-object generator. *In: Proceedings and Additional Papers, C++ Workshop.* USENIX. – San Francisco, apr 1990. To appear.

[Habert 1989] Habert (Sabine). – *Gestion d'Objets et Migration dans les Systèmes Répartis.* – Paris (France), Thèse de doctorat, Université Paris VI, décembre 1989.

[Habert 1990] Habert (Sabine), Mosseri (Laurence) et Abrossimov (Vadim). – *COOL: Kernel Support for Object-Oriented Environments.* – Rapport technique n° 1211, rocquencourt, INRIA, avril 1990.

[Jones 1985] Jones (M. B.), Rashid (R. F.) et Thomson (M. R.). – Matchmaker: an interface specification language for distributed processing. *In : Proc. 12th Annual ACM Symposium on Principles of Programming Languages*. ACM, pp. 225–235. – New Orleans LA (USA), janvier 1985.

[Joseph 1988] Joseph (Thomas A.) et Birman (Kenneth P.). – *Reliable Broadcast Protocols*. – Rapport technique n° TR 88-918, Ithaca, New York (USA), Dept. of Comp. Sc., Cornell University, juin 1988.

[Jul 1988] Jul (Eric), Levy (Henry), Hutchinson (Norman) et Black (Andrew). – Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, vol. 6, n° 1, février 1988, pp. 109–133.

[Legatheaux Martins 1987] Legatheaux Martins (José). – The design of the Chorus inter-process communication facility. *In : IBERICOM'87*. – Lisbon (Portugal), mai 1987.

[Le Narzul 1989] Le Narzul (Jean-Pierre) et Shapiro (Marc). – Un service de nommage pour un système à objets répartis. *In: Actes Convention Unix 89*. AFUU, pp. 73–82. – Paris, mars 1989.

[Makpangou 1988] Makpangou (Mesaac Mounchili). – Invocations d'objets distants dans SOS. *In . De Nouvelles Architectures pour les Communications*, éd. par Pujolle (Guy). pp. 195–201. – Paris (France), octobre 1988

[Makpangou 1989] Makpangou (Mesaac Mounchili). – *Protocoles de communication et programmation par objets : l'exemple de SOS*. – Paris (France), Thèse de doctorat, Université Paris VI, février 1989.

[McKendry 1985] McKendry (Martin S.). – Clouds: a fault-tolerant distributed operating system. *IEEE Tech. Com. Distributed Processing Newsletter*, vol. SI-2, n° 6, juin 1985.

[Meyer 1987] Meyer (Bertrand). – Reusability: The case for object-oriented design. *IEEE Software*, mars 1987, pp. 50–63.

[Nelson 1981] Nelson (Bruce J.). – *Remote Procedure Call*. – Thèse de doctorat, Carnegie-Mellon University, mai 1981.

[Presotto 1988] Presotto (David Leo). – Plan 9 from Bell Labs – the network. *In : EUUG Spring '88*. EUUG, pp. 15–21. – London, avril 1988.

[Shapiro 1986] Shapiro (Marc). – Structure and encapsulation in distributed systems: the Proxy Principle. *In : Proc. 6th Intl. Conf. on Distributed Computing Systems*. IEEE, pp. 198–204. – Cambridge, Mass. (USA), mai 1986.

[Shapiro 1989] Shapiro (Marc), Gourhant (Yvon), Habert (Sabine), Mosseri (Laurence), Ruffin (Michel) et Valot (Céline). – *SOS: An Object-Oriented Operating System — Assessment and Perspectives*. – Note technique n° SOR-68, Rocquencourt (France), Projet SOR, INRIA, septembre 1989. Accepte' pour publication à "Computing Systems".

[SOR 1988] SOR. – *Programmer's Manual for SOS Prototype-Version 4*. – Rapport Technique n° 103, Rocquencourt (France), INRIA, décembre 1988.

[Stroustrup 1982] Stroustrup (Bjarne). – *A set of C classes for co-routine style programming*. – Rapport technique n° CSRT 90, Murray Hill NJ (USA), ATT, 1982.

[Stroustrup 1985] Stroustrup (Bjarne). – *The C++ Programming Language*. – Addison Wesley, 1985.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique