ParObj : un noyau de système parallèle à objets

Francois Menneteau

THESE

présentée par

François MENNETEAU

pour obtenir le grade de

DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE de GRENOBLE

(Arrêté ministériel du 30 Mars 1992)

Spécialité : Informatique

ParObj :
Un Noyau de Système Parallèle à Objets

Date de soutenance : 21 Octobre 1993

Composition du jury :

Président :	Yves	CHIARAMELLA
Rapporteurs :	Jean	BEZIVIN
	Claude	BOKSENBAUM
Examinateurs :	Xavier	ROUSSET de PINA
	Traian	MUNTEAN
Invité :	Gul	AGHA

Thèse préparée au sein du Laboratoire de Genie Informatique

Remerciements

Je tiens en premier lieu à remercier Yves Chiaramella, Directeur du LGI, pour me faire l'honneur de présider mon jury de thèse, et surtout pour m'avoir soutenu et encouragé lorsque je me trouvais au creux d'une vague un peu plus profonde que d'habitude.

Jean Bézivin, ensuite, Professeur à l'Université de Nantes, ainsi que Claude Boksenbaum, Professeur à l'université de Montpellier II, pour avoir accepté d'être rapporteurs de ce travail. Par leur lecture attentive de mon manuscrit et leurs remarques avisées, ils ont largement contribué à son amélioration.

Xavier Rousset de Pina, Professeur à l'ENSERG, pour avoir bien voulu faire parti des membres de mon jury, pour l'analyse qu'il a porté à mon travail, et enfin pour les nombreux cours "lumineux" distillés à l'ENSIMAG sur les systèmes d'exploitation.

Traian Muntean, pour avoir eu confiance en moi en m'acceptant au sein de son équipe, pour ce domaine d'étude merveilleux qu'est le parallélisme, et pour la gestion de l'équipe comme une véritable entreprise, tout en restant un "patron" libéral et sincère.

Je n'oublie pas bien sur, chacun des membres de l'équipe SYstème Massivement PArallèles : Philippe Waille, mon "cher" collègue de bureau et dépanneur occasionnel, Léon Mugwaneza, "marathonien" dans tous les sens du terme, et Pierre Bessière, pour sa disponibilité. Il y a aussi Ghazali Talbi, Harold Castro, Robert Despons, Leila Baccouche, et Ahmed Elleuch, qui à un moment ou à un autre m'ont aidé lors de la réalisation de ce manuscrit. Je remercie aussi Martine Pernice, notre dévouée secrétaire.

Silvie Giancone, qui durant six mois, m'a permis d'avoir une interlocutrice "éclairée" avec qui j'ai pu aborder des sujets plus sérieux comme l'art et les voyages (!) et avec qui j'ai pris grand plaisir à travailler.

Je remercie aussi, mon vaillant SE/30 (et l'environnement Macintosh), sans qui l'écriture de cette thèse eut été certainement bien plus pénible, et surtout le temps, particulièrement exécrable en ce début d'été 92, et qui m'a obligé à rester chez moi à rédiger ma thèse, au lieu de me promener dans Grenoble.

Je remercie mes parents, qui en me poussant à faire des études, m'ont permis d'arriver à ce stade. Même si c'est toujours difficile à reconnaître, c'est merveilleux d'avoir un père et une mère sur qui vous pouvez compter. Mes deux sœurs enfin, qui m'ont apporté, par leur manière si féminine de voir le monde, ces petits plus qu'un fils unique ne peut connaître.

Cette thèse est dédiée
Aux thésards de tous bords
Qui derrière leurs écrans
ou leurs bouts de papier
Vont épuiser leur corps
leur âme et leur patience
Pour qu'à pas de géants
Progresse cette Science…

Résumé

Le travail présenté dans cette thèse consiste à définir les fonctionnalités d'une machine virtuelle ParObj, supportant la notion d'objets concurrents et adaptée aux exigences du parallélisme massif. Cette thèse s'inscrit dans le cadre du projet PARX de l'équipe "SYstèmes Massivement PArallèles" du LGI qui vise à spécifier et à réaliser un système d'exploitation pour machines parallèles.

A travers l'analyse de quelques Systèmes Distribués à Objets connus, nous dégageons les mécanismes de base que doit supporter ParObj. Nous avons arrêté notre étude sur les aspects suivants : structures des entités, gestion des entités, gestion des interactions entre entités, et gestion des ressources.

Dans notre approche, nous offrons dans ParObj un support parallèle pour des objets passifs et actifs qui peuvent être à la fois à gros grains (fichier, processus, etc.), et à grains intermédiaires (liste chaînée, thread, etc.). Pour une gestion encore plus fine du parallélisme, nous supportons aussi la notion d'objet distribué fractionné. En revanche, nous avons décidé de laisser aux compilateurs le soin de gérer les objets à grains fins. De plus, pour éliminer les conflits d'accès aux données, nous offrons un mécanisme de synchronisation des objets.

L'architecture générale de ParObj est basée sur le modèle original à trois niveaux de processus de PARX : le thread (qui est un flot de contrôle séquentiel à l'intérieur d'une tâche), la tâche (qui est un contexte d'exécution), et la Ptâche (qui représente un programme parallèle à l'exécution). Une Ptâche définit un domaine de communication et de protection, et assure la correction sémantique du programme parallèle (synchronisation des tâches, contrôle des protocoles d'échanges, etc.). Au sein d'une Ptâche, la protection des objets est assurée grâce à des capacités. La localisation d'une entité (qui dépend de sa visibilité et de sa référence) est réalisée grâce à un mécanisme original de désignation. Les expérimentations que nous avons réalisées montrent que ce mécanisme est parfaitement adapté à la gestion du parallélisme massif.

Mots Clés : Parallélisme, Système d'Exploitation Parallèle, Système distribué à Objets, Machine Virtuelle, Micro-Noyau, Objet.

Abstract

The objective of this thesis is to defined the functionalities of a virtual machine called ParObj, that support the notion of concurrent objects, and which is suitable for scalable parallelism. This work is in keeping with the general pattern of the PARX project within the "Massively Parallel System" team of the LGI, which aims at defining and implementing an operating system for parallel machines.

By analysing of some known Object-based Distributed System, we extract the basic mechanisms that must be supported within ParObj. We focus our study on the following aspects: entities structures, entities management, interactions between entities management, and resources management.

In our approach, we offer within ParObj a parallel support for passive and active objects that can be both large grain (file, process, etc.), and medium grain (linked list, thread, etc.) objects. For an even more subtle management of the parallelism, we support the notion of fragmented objects. On the other hand, we have decided to let compilers handle furthermore fine grain objects. In order to reduce data access conflicts, we provide synchronization mechanism between objects.

The general architecture of ParObj is based on the PARX original three level parallel process model: the thread (which is a sequential flow of control within a task), the task (which is an execution context), and the Ptask (which represents a parallel program in execution). A Ptask defines a communication and protection domain, and guarantees the semantic correctness of a parallel program (tasks synchronisation, control of the communication protocols, etc.) Within a Ptask, object protection is enforced through capabilities. Entity localisation (which depends on its visibility and reference) is achieved through an original naming mechanism. The experiments we realised show that this mechanism is perfectly suited for scalable parallelism management.

Key words: Parallelism, Parallel Operating System, Object-based Distributed System, Virtual Machine, Micro-Kernel, Object.

Table des Matières

Liste des illustrations

I. Motivations

Chapitre 1 : Introduction

1.1 Programmation des machines parallèles

Le besoin croissant en puissance de calcul et les contraintes physiques, technologiques (en ce qui concerne l'intégration) et économiques (pour les coûts de fabrication) du modèle d'architecture de machines séquentielles de Von Neumann, ont amené les concepteurs de machines à penser à de nouvelles architectures. Dans cette course vers la puissance de calcul, le développement de machines parallèles a pris un essor considérable.

Parmi les différentes catégories de machines parallèles [TW91], nous nous intéressons plus spécifiquement aux machines parallèles MIMD (Multiple Instruction Multiple Data) *asynchrone*s et qui ont les caractéristiques suivantes :

- chaque processeur représente une machine de type Von Neumann autonome,
- chaque processeur exécute séquentiellement son propre flot d'instructions indépendamment des autres processeurs,
- l'échange de l'information se fait par envoi de messages,
- la synchronisation de chaque processeur doit être spécifiée explicitement..

Les machines parallèles commencent à être utilisées couramment de nos jours dans différents domaines d'application, et de gros efforts de recherches sur des environnements de programmations parallèles et des systèmes d'exploitation distribués pour ce type d'architecture sont en cours [FSDL83], [Ra86], [Bu87], [Li88], [BN89].

Mais, alors que des langages parallèles tels que CSP [Ho85], OCCAM [MS87], Linda [CG89b], Guide [De90], Orca [BKT92], etc. ont été étudiés et développés durant ces dix dernières années, il n'existe encore qu'un tout petit nombre de vrais systèmes d'exploitation explicitement pensés pour machines à fort degré de parallélisme. Citons cependant PEACE [Sc90], EDS [IB90], Helios [Ga87], Idris [Ki88], et Trollius [Br88].

Les systèmes d'exploitations classiques ne gèrent pas les problèmes relatifs au parallélisme d'une manière satisfaisante pour des applications parallèles. Le parallélisme nécessite un support plus spécifique et plus efficace (pour la gestion des ressources, la construction de mécanismes de communication, le contrôle de l'exécution, etc.) que l'exécution pseudo-parallèle dans les environnements classiques multi-tâches.

Alors que la majorité des utilisateurs ne se soucient pas de l'architecture de la machine parallèle sur laquelle leurs applications s'exécutent, certains veulent profiter de cette connaissance pour développer leurs programmes en tenant compte des spécificités de la machine. C'est pourquoi, le contrôle du parallélisme implique le développement de nouveaux services et de supports adéquats pour sa gestion.

Un système d'exploitation non spécialisé pour machines parallèles sans mémoire commune et capable de fournir de hautes performances pour des applications parallèles, nécessite deux fonctionnalités (parfois opposées). La première consiste en la définition d'un support adéquat et efficace pour les différents grains de parallélisme et de communication. La seconde réside dans le besoin de fournir à l'utilisateur une compatibilité avec les standards existants (systèmes d'exploitation et environnements de programmation en particulier), dans le but de réutiliser les logiciels existants et de réduire les problèmes de portage.

L'élaboration d'un système d'exploitation parallèle passe par la définition d'un **modèle d'exécution** efficace pour la gestion du parallélisme, c'est à dire passe par la spécification des entités gérées par le système et des différentes interactions entre celles-ci. Nous utilisons ici le terme entité pour désigner aussi bien un processus (à la Unix[1] [Ba86] par exemple), qu'un canal de communication, ou encore un simple objet.

Cependant le modèle d'exécution dépend aussi fortement du **modèle de programmation** choisi pour le développement des applications parallèles, c'est à dire l'ensemble des entités et des opérations sur ces entités. En effet, si le support pour l'écriture d'applications distribuées n'utilise que des appels du noyau par l'intermédiaire de bibliothèques, un certain nombre de problèmes peuvent apparaître, comme par exemple lorsqu'il s'agit d'envoyer à un processus distant un message contenant une structure complexe. Comme le système d'exploitation n'a pas la connaissance de l'organisation de cette structure de données, le programmeur doit écrire du code explicite de conversion de la structure en séquences d'octets lors de l'envoi du message, et de reconstruction de la structure originale lors de la réception du message, perdant au passage les bénéfices de la *vérification statique de types*, qui permet de contrôler, à la compilation, que les données envoyées et reçues sont du même format.

L'utilisation d'un langage spécifiquement conçu pour la programmation distribuée permet d'automatiser cette conversion et le contrôle de type. D'autre part, cela permet d'augmenter la lisibilité et la portabilité des applications. Enfin, et plus important encore, un langage pourra présenter un modèle de programmation qui est d'un niveau supérieur et plus abstrait que le simple modèle par envoi de messages supporté par la majorité des systèmes d'exploitation.

Mais choisir un modèle de programmation et d'exécution efficace est une tâche difficile. Combien de systèmes "bien pensés" à leur époque sont devenus inefficaces ou bien moins performants lorsque de nouvelles fonctionnalités dues à l'évolution des technologies et des matériels leurs furent ajoutées.

---

[1] UNIX est une marque déposée des laboratoires BELL d'AT&T.

Ainsi l'introduction de la notion de machine virtuelle a entraîné la refonte complète du système d'exploitation OS/360 d'IBM pour finalement donner OS/370, l'introduction de la distribution et des réseaux) sous Unix a entraîné la création de versions mieux structurées et mieux pensées comme Chorus [AGHR89], Mach [BBBC88], Minix [Ta89] ou encore OSF/1 [BBCF92]. Ceci est d'autant plus vrai en ce qui concerne le parallélisme et les machines parallèles, où l'évolution (la révolution pourrait-on dire) est encore plus rapide.

De nombreuses études ont ainsi été réalisées dans le monde, et dont l'objectif est la prévision de l'évolution de la programmation parallèle dans les dix prochaines années. Plusieurs auteurs [NBW87], [CK91], [HE91] s'accordent sur les caractéristiques suivantes :

1. Parce que les réseaux hétérogènes de stations de travail et de serveurs seront très courants, la programmation système sera hétérogène. Par conséquent les développeurs seront capable de regrouper des programmes de différents types, issus de différents langages, dans des entités uniques.

2. Pour des raisons de réutilisabilité du code, les programmeurs se tourneront de plus en plus vers des programmes déclaratifs de plus haut niveau et l'utilisation de caractéristiques des langages à objets comme par exemple le mécanisme d'héritage et la surcharge d'opérateur.

3. Il y aura de plus en plus de systèmes qui laisseront manipuler différents types de larges structures de données comme des opérations atomiques, garantissant ainsi la sérialisation des accès et la cohérence des données. Le parallélisme inhérent à la manipulation de ces structures sera implicite plutôt qu'explicite.

4. Les entrées/sorties en parallèle et le besoin de gérer des réseaux hétérogènes donneront aux systèmes d'exploitation plus de fonctionnalités, comme la gestion de la création des processus (pour, par exemple, créer les processus gourmands en entrées/sorties sur des processeurs munis des périphériques adéquats) ou l'équilibrage de charge (pour tenir compte des spécificités des processeurs).

5. Les futurs systèmes d'exploitation seront caractérisés par leur support intégral de la distribution, et par des aspects plus qualitatifs, tels que des environnements de programmation et des interfaces homme/machine plus sophistiquées.

6. On utilisera des méthodes de preuves et de tests de programme évitant les difficultés du déboggage parallèle.

7. Les systèmes réactifs et par conséquent les programmes réactifs seront omniprésents (un processus réactif est dans l'état soit actif soit en attente, son état dépendant de l'état de la file des messages associés à ce processus).

1.2 Le projet PARX

Aujourd'hui, de nombreux projets dans le monde ont commencé à s'attaquer vraiment à la réalisation de ces projections.

Dans l'équipe "SYstèmes Massivement PArallèles" du LGI a été définie ces dernières années une architecture nouvelle pour un noyau de système d'exploitation parallèle appelé PARX [LM88], [BFFG89], [La91], [SuII91], avec comme application directe, un système pour les machines Supernode [TW91], [Wa90] (architecture massivement parallèle reconfigurable de transputeurs).

L'objectif de ce projet est la réalisation d'un noyau de système d'exploitation pour machines parallèles à communication par échanges de messages. Le noyau est structuré comme un ensemble de machines virtuelles, de telle sorte que des systèmes d'exploitations traditionnels puissent être construits par dessus (sous-systèmes de ce noyau) et que de nombreuses architectures de machines puissent être supportées.

Ce noyau sert de brique de base pour la construction du système d'exploitation PAROS qui est dédié au support efficace d'applications parallèles. Il est découpé en différentes couches qui fournissent à l'utilisateur des supports d'exécutions pour divers modèles de programmation parallèles.

Le développement de PARX couvre un certain nombre de domaines comme la conception et le développement d'applications parallèles, la compilation d'extensions de langage parallèle, l'extraction d'informations utilisées par le système, la définition de nouveaux formalismes pour le parallélisme, le support à l'exécution (résolution entre autre, des problèmes d'allocation et de migration des objets, des problèmes de routage correct des messages, etc.), et la mise au point de programmes parallèles. Les mécanismes de base de PARX font l'objet de nombreuses recherches [Eu90], [TM90], [La91], [MMS91], [El92], etc. au sein de l'équipe.

Les points 3, 4 et 5 développés dans le paragraphe précédent confortent bien l'approche micro-noyau choisie pour PARX.

Or, la taille croissante des programmes et les problèmes de plus en plus ardus de maintenance des applications ont mis en évidence la nécessité de promouvoir l'abstraction des données, la modularité et la réutilisabilité des logiciels. Dans ce but, une multitude de langages ont vu le jour, chacun tentant à sa manière d'apporter une solution satisfaisante. Parmi eux se détachent les *langages à objets*[2] [MNCL89]. Ces langages de haut niveau permettent de répondre élégamment aux problèmes soulevés par les points 1, 2 et 6 du paragraphe précédent.

---

[2] Nous utiliserons ce terme pour désigner indifféremment les langages basés sur des objets comme CLU ou Orca, et les langages orientés objets comme C++ ou Smalltalk (Cf. Définitions).

Un *objet*, en effet, est une entité qui contient un état ou des données (*champs*) privées, et un ensemble d'opérations ou de procédures (*méthodes*) qui manipulent ces données. En général, l'état et le contenu d'un objet est complètement protégé vis-à-vis des autres objets. La seule manière d'examiner ou de modifier un objet, c'est de faire une requête ou d'invoquer une des méthodes de l'objet. Ceci permet de structurer fortement les applications en créant une interface bien définie pour chaque objet, tout en cachant son implantation interne.

Or un micro-noyau d'un système d'exploitation distribué est amené à être réparti sur tous les processeurs de la machine où le système est utilisé (ceci afin d'assurer la bonne gestion des communication, des processus, etc.). D'autre part, les services de base (chargeur, gestionnaire de fichiers, serveurs de nom, etc.) sont eux aussi répartis sur de nombreux processeurs (par exemple, pour des raisons évidentes de performances, le SGF pourra n'occuper que les processeurs voisins des disques). Ainsi le noyau (et aussi un grand nombre d'applications parallèles) se retrouvent découpés en un certain nombre d'entités hétérogènes, communiquant entre elles, et disposées non uniformément sur les processeurs de la machine. C'est pourquoi, encapsuler ces structures par des objets, et n'autoriser leur modification que par l'intermédiaire d'une interface bien définie (ensemble de *méthodes*), permet de résoudre élégamment et relativement simplement les interactions entre ces entités.

Au dessus du micro-noyau PARX, nous voulons donc offrir une machine virtuelle qui offre à la fois un support efficace pour la gestion du parallélisme massif et pour *la programmation objets*, de telle sorte qu'elle permette :

- à l'utilisateur d'appréhender plus facilement les concepts du parallélisme, en fournissant des abstractions de base définies au niveau du langage (processus, objet, port, etc.),

- grâce à des constructeurs du langage, de communiquer au système des informations telles que les graphes de placement des entités de son programme sur une machine cible, le nombre de processeurs minimum au deçà duquel l'exécution de son application n'est pas souhaitable, etc.,

- de construire un programme parallèle efficace (même si le programmeur n'a pas utilisé toutes les fonctionnalités du langage mis à sa disposition), grâce notamment à des outils de parallélisation automatique des applications.

- de fournir un certain nombre de services de base comme un mécanisme de désignation uniforme et transparent, un gestionnaire de fichiers réparti, une mémoire virtuelle distribuée, un service uniforme d'acheminement des messages, etc.,

- de placer de manière optimale les applications parallèles sur les différents processeurs de la machine,

- et de gérer les entités à l'exécution et leurs interactions (problème de création dynamique d'entité, d'équilibrage de charge, de migration, etc.).

Un tel système, qui allie les avantages du système distribué à la puissance de la programmation Objet, s'appelle un *Système Distribué à Objets* (où SDO en abrégé). Un SDO fournit un environnement de programmation dans lequel les applications sont constituées d'un ensemble de modules qui s'exécutent en parallèle sur un ensemble de processeurs.

L'objectif de cette thèse cependant, n'est pas de réaliser un SDO dans sa totalité, mais plutôt d'offrir un certain nombre de mécanismes de base qui permettront de faire du *parallélisme massif à objets*, qui s'exprimera à travers un nouveau modèle de programmation, appelé *modèle hybride*. Ce modèle en effet permet à la fois une programmation traditionnelle basée sur le modèle de processus à trois niveaux de PARX, et à la fois une programmation objet, de telle sorte que l'utilisateur puisse choisir le paradigme de programmation le plus approprié pour le développement de son application.

Pour atteindre cet objectif, nous allons construire, au dessus du noyau minimal de PARX (π-noyau), une machine virtuelle appelée **ParObj**, Cf. figure 1.1, qui va fournir les fonctionnalités indispensables pour la gestion des objets et du parallélisme massif (structure et gestion des objets, *transparence de localisation* et *uniformité des accès* pour les entités du système, concurrence tant au niveau des programmes parallèles que des entités qui les constituent, communication entre programmes parallèles, et chargement efficace d'applications parallèles), et qui pourra par exemple servir à l'implantation de couche d'un niveau supérieur telles que, par exemple, des langages parallèles à objets.

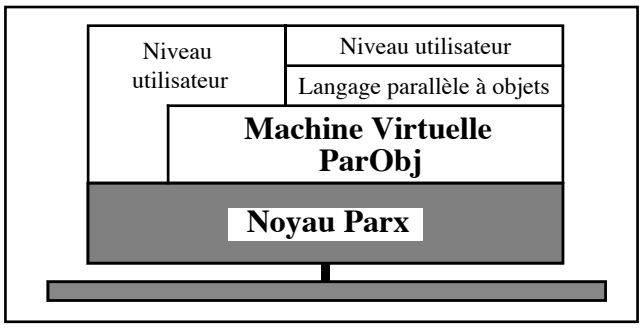


Figure 1.1. La machine virtuelle ParObj

Nous ajouterons que le travail pratique de cette thèse, ne constitue qu'une des briques de ce système parallèle, qui est une sorte de mécano évolutif, dans lequel viennent s'intégrer les différentes pièces en cours de construction dans l'équipe ([La91], [Go91], placement dynamique de processus et l'équilibrage de charge [TM92], migration de processus [El92], mémoire virtuelle distribuée [Ca91] et système de gestion de fichiers réparti [BM89], etc.).

1.3 Organisation de la suite de cette thèse

Ce manuscrit est divisé en trois parties. La première partie, qui comprend les chapitre 1 (la présente introduction) et 2, décrit les motivations de cette thèse et permet de dégager, à travers l'analyse de quelques SDO connus, les mécanismes de base que doit supporter ParObj pour la gestion des objets. La deuxième partie, qui inclut les chapitres 3 à 5, décrit les fonctionnalités de la machine virtuelle ParObj. La dernière partie enfin (chapitres 6 et 7), présente les travaux réalisés et conclut cette thèse.

Détaillons maintenant un peu plus le contenu de chaque chapitre.

Le chapitre 2 présente succinctement la notion de programmation objet, détaille les caractéristiques communément fournies par les systèmes distribués à objets, et grâce à l'étude des SDO suivants : Amœba, Argus, Clouds, COOL, Eden, Emerald, Guide, PEACE et SOS, permet de définir les concepts fondamentaux à offrir pour la structure des objets utilisés, leur gestion, les interactions entre ces objets, et la gestion des ressources.

Le chapitre 3 décrit brièvement les fonctionnalités du noyau PARX sur lesquels notre système s'appuie. PARX est en particulier basé sur modèle original de processus à trois niveaux, il offre des mécanismes génériques de construction de protocoles de communication, il permet la gestion des processeurs de la machine, etc.

Le chapitre 4 est consacré à la description de la partie "objet" de ParObj. Il contient en effet, la définition des différents classe d'objets qui seront utilisés dans PARX, leur visibilité, leur comportement et leur interaction vis-à-vis des autres entités du système (processus, objet, etc.). Il décrit aussi un moyen de sérialiser les accès aux objets et de garantir leur cohérence, grâce à la construction de *scripts de synchronisation* attachés aux objets.

Le chapitre 5 porte sur la partie "modèle d'exécution" de ParObj. Cette partie détaille comment nous avons résolu les problèmes de désignation et de protection des entités, les problèmes de chargement des applications parallèles, de leur contrôle, et de leur exécution correcte. En particulier, un mécanisme original de désignation, qui garantit à la fois l'unicité de l'identificateur (ce qui évite les conflits d'identificateurs en cas de migration par exemple), la transparence de localisation (ce qui permet d'accéder à une entité sans connaître sa localisation physique) et l'uniformité des accès (un processus est désigné de la même manière qu'un objet) est décrit.

Le chapitre 6 présente les travaux de réalisation pratique effectués, en l'occurrence l'implantation de la désignation. Les jeux de tests réalisés démontrent que le mécanisme de désignation choisi est parfaitement adapté à des architectures massivement parallèles.

Le chapitre 7 enfin, conclut ce manuscrit, et propose quelques axes de recherche pour de futurs travaux. En particulier, nous proposons la réalisation d'un langage parallèle à objets qui permettrait de faciliter le développement d'applications parallèles, tant au niveau du contrôle de leur exécution, que de l'expression du parallélism.

Chapitre 2 : Les systèmes distribués à objets

2.1 La programmation objet

La complexité et la taille des applications, aussi bien en moyens humains que matériels, nécessitent de nombreux outils de programmation perfectionnés pour amener les applications à leur terme dans les meilleurs conditions. Il est fortement souhaitable que la plus grande part possible du travail à effectuer soit réalisée par la machine, afin que les programmes soient faciles à tester, à améliorer, à réutiliser et à maintenir.

Il est bien sûr utopique de vouloir qu'un langage et son environnement de programmation satisfassent tous les besoins. Idéalement cependant, un développeur d'applications devrait pouvoir disposer des possibilités suivantes :

- Des outils d'aide à la création, la manipulation et l'exploitation de nombreux types de données.

- Des environnements de programmation perfectionnés, conviviaux (interface du système d'exploitation Macintosh [Apple88] par exemple), extensibles et configurables de tels sorte qu'ils puissent s'adapter aux besoins spécifiques de chaque utilisateur.

- Des outils pour le prototypage rapide, comprenant notamment des bibliothèques adaptées à différents types d'applications, et des outils de déboggage performants pour l'aide à la mise au point des programmes.

- Des environnement acceptant différents styles de programmation qui permettent de répondre le plus efficacement aux types de problèmes rencontrés.

- Des outils de manipulation et d'exploitation de bases de connaissances ou de données garantissant un maximum de sécurité et de cohérence.

- Des outils spécifiquement pensés pour la programmation parallèle, qui permettent à des tâches réparties sur des processeurs différents de communiquer entre elles.

A l'heure actuelle, les langages à objets constituent probablement la solution la plus souple et la plus prometteuse pour résoudre ces différents besoins, bien qu'ils ne possèdent que rarement des constructeurs spécifiquement conçus pour le parallélisme.

2.2 Définitions

2.2.1 Les objets

Un *objet* est une entité qui contient un état ou des données (*champs*) privées, et un ensemble d'opérations ou de procédures (*méthodes*) qui manipulent ces données (Cf. figure 2.1). En général, l'état et le contenu d'un objet est complètement protégé des autres objets. La seule manière d'examiner ou de modifier un objet, c'est d'exécuter une des méthodes de l'objet.

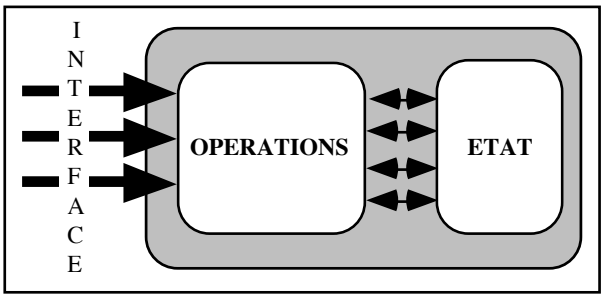


Figure 2.1. Représentation d'un objet.

Cette manière de programmer, permet de créer une interface bien définie pour chaque objet, en spécifiant les opérations possibles sur cet objet, tout en cachant la manière dont sont implantés les méthodes et les données privées.

Une méthode peut invoquer d'autres méthodes, soit du même objet, soit d'autres objets. Ces opérations peuvent à leur tour en invoquer d'autres, et ainsi de suite. cette chaîne d'invocations successives est appelée une *action* [CC91].

2.2.2 Les langages à objets

Une *classe* est l'extension directe de la notion de type abstrait. C'est le cadre à partir duquel les objets peuvent être créés. Chaque objet est une *instance* d'une classe donnée. Un groupe d'objets qui ont le même ensemble d'opérations et la même représentation de leur état interne sont considérés comme appartenant à la même classe. Typiquement, une classe existe à la compilation ; un objet existe à l'exécution.

Un langage de programmation est défini comme étant *basé sur des objets* s'il supporte les objets comme une caractéristique du langage (ex. Ada [Ada83], Modula-2 [Wi84], Orca [BKT92]) et *orienté objet* (ex. Smaltalk [GB83], C++ [St87], Sather [Om89], etc.) s'il supporte en plus le concept d'*héritage*. L'héritage est un mécanisme qui permet de définir de nouvelles classes à partir de classes existantes simplement en spécifiant les différences entre la nouvelle classe et la classe originale.

Une classe peut hériter du comportement d'une *classe de base* ou *super-classe*, et peut elle-même servir de classe de base pour une *classe dérivée* ou *sous-classe*. Il est possible d'avoir plusieurs classes dérivées d'une classe de base. Une super-classe fournit les fonctionnalités communes (comportement) à toutes ses sous-classes, tandis qu'une sous classe permet de spécialiser son comportement en ajoutant des fonctionnalités [Br85a], [Br85b], [Ca84], [Du86].

Le mécanisme d'héritage est doublement avantageux. Premièrement, en modifiant un attribut d'une classe de base (on dit en le surchargeant), cet attribut est pareillement changé dans toute ses classes dérivées et dans toutes leurs sous-classes. Deuxièmement, il encourage la *réutilisabilité* de code existant. Lorsqu'une classe n'hérite que d'une seule classe, il y a *héritage simple* (ex. Simula [DN66]). Lorsqu'une classe peut hériter de plusieurs classes, il y a *héritage multiple* (ex. C++, à partir de 1987). Une autre alternative du mécanisme d'héritage est la *délégation*. La délégation est un mécanisme qui permet à un objet d'invoquer, de manière asynchrone, un autre objet (dont il détient la référence) pour traiter la requête en cours. Ce mécanisme est par exemple utilisé dans Hybrid [Ni87] qui est un langage d'acteurs [Ag86], [AH87], et Objective-C [Lo91].

Les langages de programmation à objets encouragent leurs utilisateurs à penser et développer un programme en terme d'une multitude d'objets autonomes et interagissant entre eux par appels de méthodes (Cf. figure 2.2).

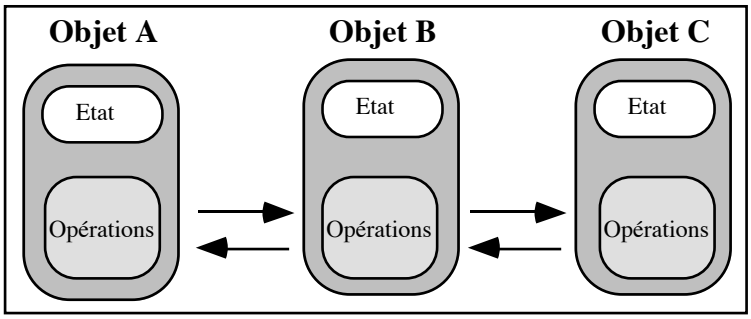


Figure 2.2. Un programme écrit en langage à objets.

Cette philosophie de diviser un programme en un ensemble de sous parties n'est pas nouvelle : c'est en fait un héritage directe de la programmation structurée des années 70 [Di71]. De cette manière de concevoir, on peut déduire les cinq caractéristiques qu'un programme à objets devrait respecter :

- **Abstraction**. La conception d'une application devrait être séparer des détails de sa réalisation. Un programme devrait cacher la manière d'effectuer les décisions et les structures de données utilisés.

- **Structuration**. Un programme est conçu comme un ensemble d'entités, avec des interactions bien définies entre les différentes entités.

- **Modularité**. La réalisation interne de chaque partie devrait être indépendante de la réalisation de toute autre partie.

- **Concision**. Le code devrait être clair et compréhensible.

- **Correction**. Un programme devrait être conçu par des méthodes qui assurent sa conformité aux spécifications et aussi être facile à tester et à déboguer.

Un langage de programmation à objets devrait être l'outil permettant une mise en œuvre avec les qualités précédemment citées. C'est pourquoi il devrait aider à la simplification de la tâche de traduction d'un problème en un programme en créant un lien beaucoup plus étroit entre le langage de programmation et l'analyse du programmeur.

2.2.3 Système à programmation par objets (SPO)

Un SPO peut être le mieux défini comme étant un environnement de développement qui fournit à la fois un langage de programmation à objet et un système qui supporte une abstraction objet au niveau du système d'exploitation. Ceci permet la manipulation, l'utilisation et la maintenance efficace des objets. Il permet aussi le partage d'objets entre plusieurs utilisateurs. Pour simplifier la gestion des objets, le système peut assigner un identificateur unique à chaque objet ("*object identifier*" ou *oid*) de telle sorte qu'il puisse être référencé d'une manière unique, et donc être retrouvé facilement.

La différence entre un système d'exploitation et le modèle de programmation qu'il supporte n'est plus aussi grande qu'il y a encore quelques années [MNCL89]. De plus en plus de fonctionnalités qui étaient l'apanage uniquement des systèmes d'exploitation sont maintenant accessibles au niveau du langage, permettant une programmation plus fine de son application (dans Mach [BBBC88] par exemple, il est possible de choisir sa propre politique de gestion des pages).

Une des raisons de ce couplage étroit réside dans le fait qu'un support efficace de bas niveau peut aussi servir pour les abstractions de plus haut niveau. Ceci fait partie du processus qui est appelé dans [NBW87] "conception d'un système total". Les auteurs pensent que lorsqu'on développe une nouvelle machine, le langage, le système d'exploitation et probablement le matériel devraient être simultanément conçus. Chaque partie peut alors être construite pour supporter un environnement particulier de telle sorte qu'un système plus uniforme et efficace est créé. De cette manière de procéder découle cependant un désavantage : le sacrifice d'une future flexibilité du système d'exploitation, si celui-ci est conçu uniquement pour des abstractions particulières.

Le système d'exploitation d'un SPO donne la vision d'un espace d'adressage global pour les objets dans toute la machine en fournissant les caractéristiques suivantes :

- **Gestion des objets**,
- **Gestion de l'interaction entre les objets,**
- **Gestion des ressources de la machine**.

2.2.4 Systèmes distribués à objets (SDO)

Nous utilisons la définition donnée dans [BST89] : *"un système distribué est constitué d'un certain nombre de processeurs autonomes sans mémoire commune, connectés entre eux par une sorte de réseau, et communiquant par échanges de messages"*. Nous incluons donc les systèmes répartis[3] et les systèmes parallèles[4].

Le développement des systèmes d'exploitation distribués et des langages de programmation basés sur des objets rendent possible un environnement dans lesquels les programmes sont constitués d'un ensemble de modules, ou d'objets qui s'exécutent en parallèle sur un ensemble de processeurs. En résumé, la programmation objet permet la création de programmes qui sont une collection d'entités interdépendantes, et les systèmes distribués permettent la vision d'un ensemble de processeurs comme une machine unique.

Typiquement, un SDO a, dans le meilleur des cas, les fonctionnalités suivantes :

- **Distribution**. Par définition.

- **Transparence**. Le système peut cacher l'environnement distribué à l'utilisateur. Par exemple, il peut garantir la *transparence*[5] *de localisation* des données de telle sorte que l'utilisateur n'a pas besoin de connaître l'emplacement physique d'un objet pour l'invoquer. Le système peut aussi fournir un accès uniforme à tous les objets, que ceux-ci soient présents en mémoire ou stockés en mémoire auxiliaire.

- **Intégrité des données**. Un SDO s'assure qu'un objet persistant est toujours dans un état cohérent avant d'effectuer une opération sur cet objet. C'est à dire qu'un objet est toujours dans un état qui est le résultat d'une opération qui s'est terminée correctement. Dans le cas contraire, le système garantit que toutes les modifications appliquées à l'objet ont été défaites.

- **Tolérance aux défaillances et Récupération**. La défaillance d'un processeur ou d'un objet constitue seulement une panne partielle dans un SDO. La perte est restreinte à ce processeur ou cet objet. Le système devrait être quand même capable de continuer l'exécution des programmes bien qu'il puisse souffrir d'une perte de performances ou de services. Il devrait aussi être capable de récupérer le ou les objets défaillants.

---

3 Réseau de machines ne partageant pas de mémoire et communiquant par échanges de messages, au moyen d'un *réseau d'interconnexions faiblement couplé (réseau local par exemple)*. Chaque machine est généralement un mini-ordinateur, un poste de travail ou une machine spécialisée.

4 Ensemble de machines autonomes ("nœuds") sans mémoire commune et communiquant par échange de messages, à travers un *réseau d'interconnexions rapide et fiable (liaisons bipoints)*. Les nœuds sont souvent fortement couplés, i.e. ils démarrent, coopèrent et s'arrêtent généralement simultanément.

5 Il serait en fait peut-être plus logique de parler *d'opacité* puisque le but de cette fonctionnalité consiste à *cacher* à l'utilisateur la localisation de l'objet.

- **Disponibilité**. Un SDO doit prendre un certain nombre de mesures pour s'assurer que, quels que soient les impondérables matériels (pannes, etc.), tous les objets restent disponibles avec une grande probabilité.

- **Protection**. Un SDO devrait permettre au possesseur d'un objet de spécifier le ou les clients qui sont susceptibles d'utiliser cet objet, ainsi que leurs droits.

- **Parallélisme au niveau du programme**. Un SDO devrait être capable de répartir les objets sur différents processeurs de telle sorte qu'ils puissent s'exécuter en parallèle (Cf. figure 2.3).

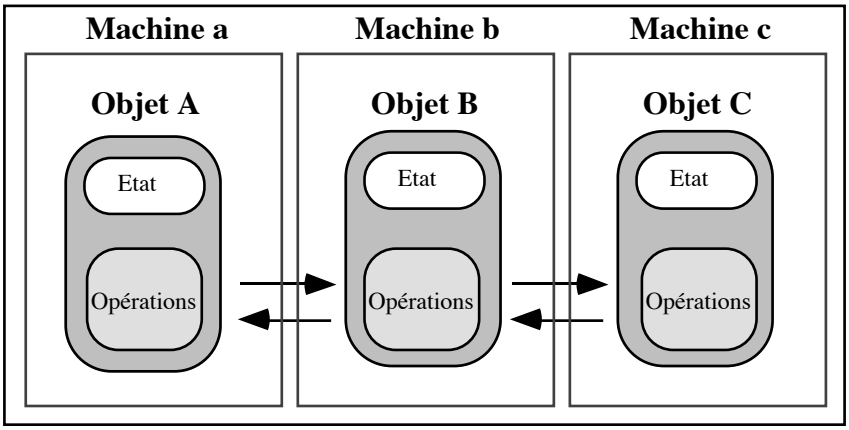


Figure 2.3. Un programme distribué écrit en langage à objets.

- **Concurrence au niveau de l'objet.** Un objet devrait être capable de satisfaire plusieurs requêtes en parallèle. Il est à noter que dans la plupart des cas, il s'agit simplement de pseudo-parallélisme, puisque les requêtes seront traitées par un même processeur.

- **Meilleures Performances**. Un programme correctement conçu devrait s'exécuter plus efficacement sur un SDO que sur un système conventionnel, à condition bien sur, de ne payer trop cher le surcoût introduit par la structuration de l'application.

2.3 Structure des objets

La structure d'un objet supporté par un SDO influence fortement l'ensemble de la conception du système. Un SDO peut être spécifié par :

- La *granularité des objets*, qui caractérise la taille relative de l'objet, la quantité d'information traitée par celui-ci et le coût introduit au niveau système pour la manipulation de cette information.

- Le *comportement des objets*, qui caractérise le type d'interaction entre les processus (exécution séquentielle, dans un espace d'adressage virtuel, d'une suite d'instructions représentant un programme) et les objets du système.

2.3.1 Granularité d'un objet

Dans le cas le plus usuel, un SDO supporte seulement des *objets à gros grains*. Ces objets sont caractérisés par leur taille importante, leur grand nombre d'instructions exécutées, et par leur faible interaction avec les autres objets du système. Comme type d'objets à gros grains on peut citer par exemple un fichier ou une base de données mono-utilisateur. Typiquement, ces objets résident dans leur propre espace d'adressage. Ceci permet de fournir une protection matérielle entre les différents objets et d'assurer qu'une erreur logicielle n'implique que l'objet fautif.

Le modèle pur à gros grain offre l'avantage de la simplicité, mais il présente aussi un certains nombre d'inconvénients :

- La gestion de ces entités est relativement lourde.
- Le contrôle et la protection des données se trouvent au niveau de l'objet (à gros grain), ce qui restreint la flexibilité du système et diminue fortement les accès concurrents (puisque l'objet est vu comme un seul bloc et non un ensemble de sous-blocs).
- Le système n'offre pas un modèle de données cohérent : les grosses entités sont des objets alors que des entités plus petites, comme des entiers ou des listes chaînées, sont représentés comme les données des langages de programmation traditionnels.

Quelques exemples de SDO à gros grains : **Amœba** [TM80] (chaque objet est composé d'un certain nombre de segments de code et de données qui peuvent être partagés avec d'autres objets), **Clouds** [Da90] (un objet est constitué d'un certain nombre de segments : code, données, tas, les segments de code pouvant être partagés par plusieurs objets. Des paginateurs matériels spécialisés sont utilisés pour placer efficacement les objets dans l'espace d'adressage des processus), et **Eden** [ABLN85] (chaque objet est constitué de trois parties : un état à longue durée de vie qui contient les informations permanentes, un état à brève durée de vie qui contient les informations volatiles, et un segment de code qui contient les opérations).

Pour fournir un niveau de contrôle plus fin sur les données, un SDO pourrait supporter éventuellement des objets à gros grains et des *objets à grains intermédiaires*. Ces derniers objets sont créés et maintenus à moindre coût (puisqu'ils sont plus petits) tant au niveau de leur taille qu'à l'étendue de leur action. Des exemples typiques d'objets à grains intermédiaires : listes chaînées, file d'attente, et les composants d'une base de données multi-utilisateur.

Un certain nombre de ces objets peuvent résider dans l'espace d'adressage d'objets à gros grains. Ceci permet à un objet à gros grain d'avoir un degré de concurrence plus grand, puisque la synchronisation des accès aux données (Cf. paragraphe **4.2.6 Synchronisation**, dans le chapitre suivant) peut se faire maintenant au niveau des objets à grains intermédiaires. De manière similaire, lors, par exemple, de l'écriture sur disque d'un objet à gros grain, le volume des données transférées sera fortement réduit, puisque seuls les objets à grain intermédiaires qui ont été modifiés seront sauvegardés.

L'inconvénient de l'utilisation d'objet à grain intermédiaire réside dans un surcoût additionnel introduit au niveau du système, car celui-ci doit maintenant gérer deux niveaux d'un bien plus grand nombre d'objets. De plus, le modèle de données n'est toujours pas complètement cohérent.

Argus [Li88] est un exemple typique de SDO qui gère des objets à gros grains et grains intermédiaires (le domaine d'exécution est le *gardien*. C'est un gros objet qui contient des objets du langage et des processus). Il existe aussi des SDO qui ne supporte que des objets à grains intermédiaires. Citons par exemple **COOL** ("Chorus Object-Oriented Layer") [HMA90], et **SOS** [SGHM89], [Ha89].

Dans COOL un objet est divisée en deux parties : une partie dans l'espace d'adressage utilisateur et qui est constituée d'un segment de code (méthodes) et d'un segment de données placés dans un contexte donné. La partie dans l'espace d'adressage système contient un descripteur qui permet de gérer l'objet (attributs, représentation, droit d'accès, etc.). Chaque objet réside dans un espace d'adressage local à un site (appelé "*context*") et dans lequel s'exécutent les threads (fournis par le noyau Chorus). Un ensemble d'attributs, associés à chaque objet, détermine si l'objet est *global* (il peut recevoir des messages d'objets distants) et/ou *permanent*.

Dans SOS, un objet s'exécute aussi dans un *contexte* ou espace d'adressage virtuel. Il faut distinguer l'*objet élémentaire* dont la représentation est localisée dans un seul contexte, de l'*objet fragmenté* qui est réalisé par un groupe d'objets élémentaires, ses fragments, éventuellement localisés dans différents contexte sur différents sites. Sa représentation est l'union de tous les fragments.

Finalement pour fournir un degré de contrôle encore plus fin sur les données, un SDO pourrait supporter des objets à gros grains, des objets à grains intermédiaires et des *objets à grains fins*. Ces derniers objets sont caractérisés par leur petite taille, le petit nombre d'instructions qu'ils exécutent, et le nombre relativement élevé d'interactions avec les autres objets. Des exemples d'objets à grain fins sont des types de données fournis par les langages de programmation conventionnels comme les booléens, les entiers, et les nombres complexes.

Des exemples typiques de SDO supportant les trois modèles : **Emerald** [BHLC87], [JLHB88], et **PEACE** [Sc90], [Sc91], [Sc92].

Dans Emerald, le noyau ne supporte cependant que les objets à gros et moyen grain. Les objets à grain fin ne sont pas vus par le noyau et sont directement incorporés dans

le code des programmes (ensemble d'objets) par le compilateur. Les objets à grain fin et à grain intermédiaire ne sont pas visibles en dehors de l'objet qui les contient. Les objets à gros grain peuvent être invoqués par n'importe quel autre objet et sont les seuls objets qui peuvent être migrés. Emerald distingue trois type d'objets : les *globaux* (visibles et accessibles par n'importe quel objet) les *locaux* (complètement contenu dans un autre objet, et non visible par l'extérieur), et les *directs* (objets locaux à grain très fin, comme les entiers par exemple).

Dans PEACE, les objets à grains fins sont aussi gérés par le compilateur. L'unité de distribution est appelée *team*. Les threads (objets actifs, Cf. paragraphe suivant) s'exécutent en concurrence dans l'espace d'adressage d'un team. Un ensemble de team peut être regroupé au sein d'une *league* fournissant un domaine de communication protégée.

Enfin, **Guide** [De90], [KMNR90] est un système et un langage qui ne supporte qu'un modèle à objets à grains intermédiaires et fins. Seul les objets identifiés par une *référence système* (i.e. les objets persistants) sont connus du système de gestion des objets. L'unité d'exécution est le *domaine* qui peut s'étendre sur plusieurs sites, et peut être vu comme une machine virtuelle multi-processeurs. Un domaine s'exécute dans un espace d'adressage unique appelé *mémoire virtuelle d'objets*. Il est composé d'un certain nombre de flots d'exécution séquentiels appelés *activité* qui s'exécutent sur un site donné.

2.3.2 Comportement d'un objet

Lorsque les processus sont séparés des objets et qu'ils sont temporairement attachés à eux lors de l'invocation de leurs méthodes, on parle d'un *modèle à objets passifs*. Si en revanche, les processus sont couplés et attachés en permanence aux objets dans lesquels ils s'exécutent, on parle d'un *modèle à objets actifs*.

Permettre à une multitude de processus de s'exécuter en concurrence à l'intérieur d'un objet, autorise le traitement en parallèle d'une multitude de requêtes. Cependant, le nombre réel d'exécutions simultanées d'objets à un moment donné dépend du type de synchronisation supporté par le système et du type de la requête reçue.

2.3.2.1 Modèle à objets passifs

Dans ce type de modèle, les processus et les objets sont donc des entités complètement séparés. Un processus n'est ni attaché ni restreint à un objet unique. En fait, un unique processus est utilisé pour satisfaire toutes les opérations nécessaires à la réalisation d'une action. Par conséquent, un processus, au cours de son existence, peut exécuter les méthodes de différents objets. Quand un processus invoque un objet (O2), son exécution à l'intérieur de l'objet courant (O1) est temporairement suspendue. Le processus est alors placé dans l'espace d'adressage de l'objet O2, et il exécute l'opération appro-

priée. Quand le processus termine cette opération, il retourne à l'objet O1, et continue l'exécution à l'endroit où il avait été interrompu (Cf. figure 2.4).

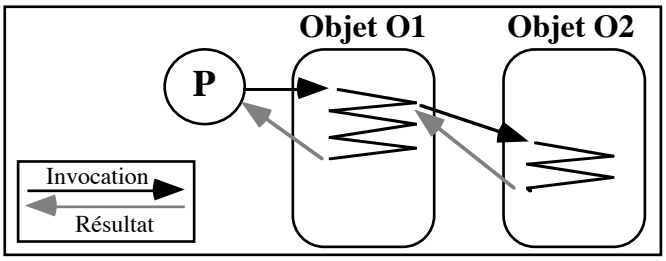


Figure 2.4. Réalisation d'un action dans le modèle d'objets passifs.

Un avantage de ce modèle réside dans le fait qu'il n'y a virtuellement pas de restrictions au nombre de processus qui peuvent invoquer un objet. Les inconvénients proviennent du coût et de la difficulté à placer et à enlever un processus de l'espace d'adressage d'un objet.

Clouds, *COOL*, *Guide* et *SOS* sont des SDO à modèle à objets passifs.

2.3.2.2 Modèle à objets actifs

Dans ce modèle, un certain nombre de processus sont crées et assignés à chaque objet pour traiter ses requêtes. La portée de chaque processus est limitée à l'objet. Lorsqu'un objet est détruit, il en est de même de tous ses processus. Dans le modèle à objets actifs, une opération n'est pas accédée directement par le processus appelant, comme dans le cas d'un appel de procédure. Quand un *client* invoque une opération, un processus dans l'objet *serveur* correspondant accepte la requête et effectue l'opération au compte du client. Les interactions entre les clients et le serveur sont les suivantes :

(1) Le client présente à l'objet désiré une requête, ainsi qu'une liste d'arguments spécifiant le type d'opération à invoquer.
(2) L'objet serveur accepte la requête, puis localise et réalise l'opération désirée.
(3) Si durant l'exécution de l'opération, une invocation d'un autre objet est nécessaire, le processus émet la requête et attend la réponse. Un processus serveur dans l'autre objet est alors appelé pour exécuter la nouvelle opération, et ainsi de suite.
(4) Quand l'opération se termine, le serveur retourne le résultat au client.

Dans cette approche de multiples processus peuvent être impliqués pour réaliser la même action (Cf. figure 2.5).

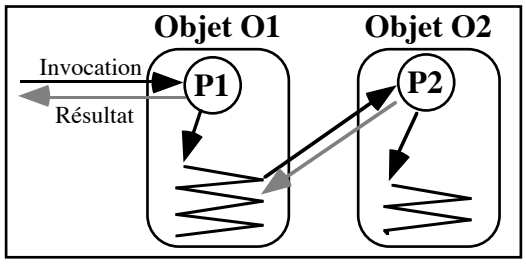


Figure 2.5. Réalisation d'une action dans le modèle d'objets actifs.

Dans la version *statique* de ce modèle, un nombre fixé de processus serveurs sont lancés pour chaque objet créé ou activé. Lorsqu'une requête est envoyée à un objet, elle est indifféremment assignée à un processus inactif, qui réalise alors l'opération demandée. Les requêtes qui sont envoyées à un objet dont les processus serveurs sont tous occupés, sont placées dans une file d'attente et seront traitées plus tard. Dans ce schéma, le degré maximum de parallélisme d'un objet est limité par le nombre de processus serveurs qui lui sont associés.

Citons comme SDO supportant ce modèle : *Amœba*, *Eden* (chaque objet contient un ensemble de processus "Dispatcher" [serveur(s) et maintenance(s)], dont le rôle est d'accepter les requêtes destinées à l'objet et de les assigner à des processus inactifs, *Emerald* (qui supporte aussi un modèle à objet passif) et *PEACE* (un seul processus par objet actif).

Dans la version *dynamique* du modèle actif, les processus serveurs d'un objet sont dynamiquement créés à la demande. Les requêtes ne sont jamais stockées dans une file d'attente. Lorsqu'un processus a terminé le traitement d'une opération, il est détruit. Dans ce schéma, le degré de parallélisme est maximal. cependant il est un peu plus coûteux, puisqu'il implique la création dynamique et la destruction de processus.

Un SDO qui supporte ce modèle : *Argus* (les processus serveurs d'un objet sont crées dynamiquement à la demande. Le coût de création et de destruction dynamique de processus est réduit en gérant une liste de processus non utilisés : tout nouveau processus créé est d'abord pris dans cette liste, tout processus détruit est remis dans cette liste. Ce n'est que lorsque la liste est vide que l'on crée un nouveau groupe de processus).

L'analyse de ces différents SDO montre que les objets à grains fins sont généralement laissés à la charge du compilateur et que les objets à grains intermédiaires présentent le meilleur compromis entre l'augmentation du degré de parallélisme d'un programme et le coût introduit pour leur gestion.

La plupart des SDO assignent à une entité (objet ou flot de contrôle) un identificateur qui est unique dans tous le système. Ceci permet d'éviter les conflits et d'autoriser des mécanismes tels que la migration des objets.

Afin d'optimiser les communications entre entités, certains SDO (Emerald, COOL) font la distinction entre objets **globaux** (qui peuvent être accédés par des entités distantes, ce qui implique une communication par envoi de messages) et objets **locaux** (qui ne sont pas visibles en dehors d'un certain contexte, ce qui permet une communication par appel de procédure).

Enfin, la gestion de la répartition a entraîné la création d'abstractions tels que les leagues (PEACE), ou les domaines (Guide) dont le rôle est d'offrir un espace de protection et/ou de communication pour les applications, et qui s'étend sur plusieurs processeurs.

De l'étude sur la structure des objets nous pensons qu'un modèle à objets complètement uniforme ne représente pas un concept fondamental. En particulier, les objets à grains fins ne devront jamais être gérés par le système, sous peine d'introduire un surcoût insurmontable pour leur gestion. Nous avons donc décidé que notre machine virtuelle devait supporter des objets à gros grains et à grains intermédiaires. Les objets à gros grains n'offrant peu de support pour les accès concurrents, nous verrons dans le chapitre suivant, comment remédier à ce problème en utilisant des objets fragmentés (dont la sémantique est différente de celle utilisée dans SOS par exemple).

D'autre part, nous pensons qu'un pur modèle à objets actifs, bien qu'offrant un degré de parallélisme maximum, devient fortement pénalisant lorsque l'architecture cible devient une machine parallèle. En effet, ce modèle assignant à chaque objet au moins un processus (pour le traitement des requêtes envoyées à l'objet), nous en déduisons que le coût introduit pour sa gestion est très nettement supérieur à un objet passif. Ceci devient particulièrement vrai dans le cas ou les objets sont très nombreux et de petite taille (cas des applications scientifiques en particulier). Enfin ce modèle présente un inconvénient non négligeable lors de la migration d'objets : il faut non seulement déplacer les données de l'objet, mais aussi le ou les processus qui lui sont rattachés, ce qui est une tâche relativement complexe [El92]. C'est pourquoi nous avons décidé d'offrir à la fois un modèle à objet passif (pour, par exemple, la manipulation de gros volumes de données, comme les matrices) et un modèle à objet actif (pour, par exemple, réaliser des serveurs de services à forte disponibilité).

2.4 Gestion des objets

Les objets sont les entités fondamentales des SDO. Par conséquent, leur gestion est une fonction essentielle de ces systèmes. Dans cette partie, nous nous intéressons aux principales techniques qui permettent :

- de *synchroniser* les exécutions concurrentes d'opérations au sein de l'objet,
- et de *protéger* les objets contre des accès non autorisés.

2.4.1 Synchronisation

Une autre caractéristique importante des SDO est de garantir que les activités de multiples actions qui impliquent le même objet, ne rentrent pas en conflit ou n'interfèrent pas les unes avec les autres. Une action ne devrait pas être autorisée à observer ou modifier l'état d'un objet qui a été partiellement modifié par une autre action qui n'a pas encore été validée. Ainsi pour garantir la sérialisation et pour protéger l'intégrité des états des objets, un mécanisme de synchronisation est nécessaire. On peut grosso modo distinguer deux grandes catégories :

- Les mécanismes de *synchronisation pessimistes***.** Dans ce cas le système ou l'utilisateur prend les mesures nécessaires pour éviter les conflits (en suspendant temporairement les actions qui vont interférer avec l'action en cours).

  Les SDO suivants supportent un mécanisme pessimiste : *Amœba* (lorsqu'une action affecte plusieurs objets), *Argus* (qui utilise des simples verrous en lecture/écriture. De plus, une action et ses sous actions ne peuvent modifier simultanément un même objet), *Clouds* (deux niveaux de mécanismes sont fournis : un mécanisme automatique (réalisé par le système) qui utilise des verrous en lecture/écriture, et un mécanisme personnalisable qui permet à l'utilisateur de créer des règles spécifiques de synchronisation en employant des sémaphores ou des verrous. Le mécanisme personnalisable a l'avantage de pouvoir augmenter le degré de parallélisme des accès. En revanche, la sérialisation des accès ne peut plus être absolument garantie par le système [cas d'une erreur de programmation par exemple]), *COOL* (comme dans Chorus, toutes les invocations qui sont faites par un processus sont déposées dans une file d'attente, et que ces requêtes ne seront traitées qu'après la terminaison de l'opération en cours, nous en déduisons que chaque objet ne peut réaliser qu'une seule invocation à la fois), *Eden* (utilisation de moniteurs au niveau du langage), *Emerald* (moniteurs), *Guide* (la sérialisation des accès aux méthodes des objets se fait grâce à des clauses de synchronisation attachées à chaque méthode et qui contiennent des variables de l'objet, des paramètres de la méthode et des compteurs de synchronisation, Cf. **Chapitre 4** sur la synchronisation des objets pour plus d'informations), *PEACE* (sémaphores), et *SOS* (verrous en lecture/écriture).

- Les mécanismes de *synchronisation optimistes*. Comme un objet ne prend aucune précaution pour prévenir les conflits, avant de valider une action, il faut s'assurer que les opérations sur les données en cours ne vont pas entrer en conflit avec des modifications déjà effectuées par une autre action (qui a été précédemment validée). Par exemple, si des données détenues sont obsolètes, alors toute mise à jour avec ces données risque d'être incorrecte. En conséquence, cette action doit être avortée. Pour réduire les conflits potentiels, les objets peuvent être représentés par un ensemble de pages ou d'objets d'un grain inférieur permettant ainsi une modification indépendante. Ceci permet un maximum de concurrence.

  En plus du mécanisme pessimiste déjà cité, *Amœba* supporte aussi un mécanisme optimiste lorsqu'un seul objet est pris en compte.

2.4.2 Protection

Fournir un mécanisme de protection sur les objets afin de garantir que seuls les utilisateurs autorisés pourront les invoquer est une nécessité. Ce mécanisme peut être fourni soit par le système soit par l'utilisateur.

Un mécanisme courant de protection est celui basé sur des *capacités*. Une capacité est une clé, souvent cryptée (pour empêcher les contrefaçons), qui contient deux champs : un nom qui identifie un objet et des droits d'accès. Chaque capacité désigne un seul objet. Cependant un objet peut avoir plusieurs capacités (ceci permet ainsi des variations au niveau des droits d'accès de l'objets). Pour pourvoir accéder à un objet, il faut détenir au moins une capacité de cet objet, et cette capacité est passée en paramètre de toute requête. La vérification, la gestion et la maintenance des capacités peut être la responsabilité soit du système, soit des objets.

Un autre mécanisme est basé sur une *procédure de contrôle*. Avec cette approche, chaque objet dispose d'une procédure spéciale qui filtre les requêtes en contrôlant par exemple la liste les utilisateurs autorisés à invoquer l'objet. Ce mécanisme est évidemment le plus flexible et très difficile à contourner, puisque les informations de protection sont contenues dans l'objet lui même.

Ces deux mécanismes sont souvent combinés pour offrir un niveau maximum de protection.

Les SDO suivant utilisent des capacités pour la protection des objets : *Amœba* (une plus grande sécurité peut encore être obtenue en cryptant ces capacités à l'aide de puces spécialisés insérées entre chaque processeur et le réseau), *Clouds*, *COOL* (pour les objets globaux), *Eden*, *PEACE*, *SOS* (pour les mandataires de chaque objet).

Peu de SDO permettent une synchronisation fine des accès. L'unité d'accès est généralement l'objet, ce qui ne permet pas un fort degré de parallélisme. Seuls Clouds et surtout Guide (sérialisation possible au niveau de la méthode) permettent une synchronisation personnalisée.

L'utilisation de capacités semble actuellement être le mécanisme le plus usité pour la protection des accès aux objets. Il s'accompagne souvent d'un cryptage des informations sensibles.

De l'étude sur leur gestion, nous pensons que la synchronisation des objets ne doit pas être seulement assurées par le système, car dans ce cas l'unité d'accès est l'objet, ce qui ne permet pas une synchronisation fine des accès (qui permet d'augmenter le degré de parallélisme). C'est pourquoi nous avons décidé d'offrir un mécanisme personnalisable basé des scripts de synchronisation attachés aux méthodes des objets. Enfin, un système devant toujours offrir un mécanisme de protection, même rudimentaire, pour garantir la sécurité des objets, nous protégerons les objets de ParObj par des capacités (ce choix ayant été imposé par le mécanisme de protection de PARX qui s'appuie sur des capacités).

2.5 Gestion des interactions entre objets

Un SDO est responsable de la gestion des invocations entre des objets coopérants. Quand une action lance une requête, le système doit localiser l'objet spécifié, prendre les mesures appropriées pour invoquer l'opération spécifiée, et éventuellement retourner un résultat. Cette partie décrit les caractéristiques fournies par un SDO pour localiser les objets serveurs, et pour gérer l'interaction entre les objets. Les méthodes de détection des invocations qui échouent ne seront pas abordées.

2.5.1 Localisation d'un objet

Un SDO devrait pouvoir fournir la propriété de *transparence de la localisation* de telle sorte qu'un client ne devrait pas se préoccuper de la localisation physique d'un objet pour pouvoir l'invoquer. A chaque fois qu'une invocation est effectuée, le système doit déterminer quel objet est invoqué et sur quel processeur il se trouve. Le mécanisme de localisation d'un objet devrait être suffisamment flexible pour permettre la migration des objets à travers le réseau.

On distingue quatre grandes techniques de localisation des objets :

- **L'encodage de la localisation de l'objet dans l'identificateur de l'objet**. Pour déterminer le processeur cible, le système analyse simplement un des champs de l'identificateur. C'est la méthode la plus efficace. Cependant, dans le cas où les identificateurs d'objet sont uniques, cette technique empêche la migration des objets, puisque changer de processeur implique un changement de l'identificateur de l'objet.

  Les SDO suivants n'utilisent que ce mécanisme pour la localisation des objets : *Argus* (à l'intérieur de chaque objet à gros grain, chaque processus reçoit un identificateur unique qui permet de le désigner lorsqu'une réponse est renvoyée après une invocation vers un autre objet), et *Guide* (lorsqu'un objet est invoqué, si l'objet n'est pas chargé en mémoire virtuelle sur le nœud où l'invocation a eu lieu, une *faute d'objet* est déclenchée. L'objet est alors localisé en mémoire secondaire grâce aux indications de localisation contenues dans sa référence système. Un nœud est choisi pour l'exécution et l'objet est chargé sur ce nœud).

- **L'utilisation de liens de poursuite de l'information**. A chaque fois qu'un objet est migré d'un processeur P1 à un processeur P2, un pointeur de relégation contenant P2 comme destination est installé sur le processeur P1. Pour retrouver un objet qui a migré, le système suit simplement les pointeurs de relégation jusqu'au processeur final. Cependant en cas de défaillance d'un processeur qui contient un pointeur de relégation, il n'est plus possible de retrouver l'objet spécifié. **Note** : ce mécanisme est le plus souvent combinée avec d'autres mécanismes (par exemple encodage de la localistion, cache/diffusion, etc.) pour augmenter les performances du système.

- **L'utilisation d'un cache et d'un mécanisme de diffusion**. Un cache est maintenu sur chaque processeur qui contient les dernières localisations connues d'un certain nombre d'objets récemment accédés à distance. S'il arrive une requête pour un objet dont la localisation n'est pas dans le cache, un message est diffusé sur le réseau, demandant la localisation de cette objet. Lorsqu'il reçoit une réponse, il met à jour son cache. Cette solution est très efficace (rapidité de la localisation d'un objet) et flexible (puisqu'elle autorise la migration d'objets). Cependant la diffusion de messages peut surcharger le réseau.

  Clouds est un exemple de SDO qui utilise uniquement ce mécanisme : lorsqu'une requête est destinée à un objet qui n'est pas local, l'objet de gestion des communications locales diffuse à tous ces homologues un message avec le type "recherche et invoque". Lorsqu'un objet de communication a trouvé l'objet (dont la capacité est spécifiée dans le message) il crée alors un processus qui va effectuer l'invocation pour le client, puis renvoyer une réponse.

  La procédure de recherche est rendue plus rapide en tenant à jour trois structures qui conservent respectivement la liste des objets actifs de la machine (*table des objets actifs*), la liste des objets qui peuvent être présents localement et ceux qui n'ont aucune chance d'y être (*table des "peut-être"*), et enfin la liste de tous les objets actifs ou inactifs de la station de travail (*répertoire des objets*). La procédure de recherche examine chaque structure successivement, ce qui permet de réaliser des tests rapides et efficace d'appartenance avant de se lancer dans la recherche de tout le contenu de la machine.

  Eden utilise des caches locaux et un mécanisme spécial de recherche pour tenir compte de l'extrême mobilité des objets.

  Amœba en revanche, combine ce mécanisme avec l'encodage de la localisation dans l'identificateur de l'objet : chaque objet est identifié par un nom global et unique (exprimé par sa capacité). L'identificateur de l'objet contient le numéro de port (codé sur 48 bits) du service qui gère l'objet, et le numéro local de l'objet. Lorsqu'un message est envoyé à un objet, il est en fait envoyé au port (donc au service) dont le numéro est contenu dans la capacité passée en paramètre de l'appel. Pour retrouver un service, Amœba maintient un cache des dernières localisations des ports, et diffuse un message lorsque une localisation est inconnue.

  Emerald, quant à lui, combine un mécanisme à caches/diffusion avec des liens de poursuite de l'information. Chaque entrée dans le cache est couplé à un horodateur qui indique l'age relatif de la donnée. Si une localisation est trouvée dans le cache mais avec une date périmée, une autre localisation plus récente est réclamée. S'il en existe une, cette localisation est testée, dans le cas contraire, une requête de localisation est diffusée (grâce à un protocole de diffusion fiable).

  SOS utilise aussi un mécanisme de cache/diffusion avec liens de poursuite au niveau des caches. Mais, en plus, il pousse le raffinement jusqu'à initialiser les

caches locaux avec les informations de localisation contenues dans l'identificateur de l'objet (appelé *référence d'objet*).

COOL enfin, combine l'encodage, les liens de poursuite, et la diffusion : chaque objet est désigné par un unique identificateur (UI). Cette identificateur comporte l'identificateur du serveur de l'objet (adresse d'un port), et l'identificateur local de l'objet (qui est serveur dépendant). La localisation d'un objet se fait en utilisant l'identificateur du port (serveur) contenu dans son UI. Celui-ci contient en effet le numéro de la machine sur laquelle le port correspondant a été originellement créé, ce qui permet d'envoyer une requête de localisation vers cette machine. Comme chaque machine tient à jour la localisation des ports qui ont été créés en son sein, elle peut donc renvoyer une réponse. Dans le cas où un port ne peut être situé, (par exemple, parce que la machine qui détient l'information est défaillante) une requête de localisation est diffusée à travers tout le réseau.

- **L'utilisation d'un serveur distribué de noms**. Le système maintient un certain nombre d'objets serveurs de noms qui coopèrent entre eux afin que collectivement ils contiennent des informations à jour sur la localisation de tous les objets du système. Un serveur peut contenir soit la totalité des localisations et dans ce cas il est capable de traiter n'importe quelle requête, soit une information partielle et dans ce cas si une requête ne peut être satisfaite, elle est routée vers un autre serveur de noms. L'inconvénient de cette solution réside dans la difficulté de conserver des informations cohérentes entre les différents serveurs.

  PEACE est de loin, le SDO qui possède le mécanisme de localisation le plus sophistiqué : chaque objet est désigné par un unique identificateur ("*unique*"). Cet identificateur est divisé en trois champs : un identificateur qui donne la localisation théorique de l'objet (numéro de machine), un identificateur de team (unité de distribution), et un identificateur local unique ("*lui*"). Chaque team est lié à un espace (appelé "*domain*") dans lequel le "lui" est unique et auquel un certain nombre de services sont associés. Il existe un gestionnaire de noms par domaine. Les domaines sont organisés de manière hiérarchique.

  La localisation d'un objet se fait en utilisant d'abord l'identificateur de localisation qui permet d'envoyer la requête vers cette machine. Celle-ci recherche alors le port local ("*gate*") correspondant au "lui" contenu dans l'identificateur et dépose le message. Dans le cas ou l'objet a migré, le port sert de lien de poursuite en routant le message vers la destination courante de l'objet. Afin de réduire les temps de réponses, PEACE utilise des caches locaux pour conserver la localisation de ports distants. Lorsque l'information de localisation n'est pas disponible, le système s'adresse au serveur de nom associé au domaine dans lequel réside le team. Si celui-ci ne peut satisfaire la requête, il route la requête vers le serveur de nom qui gère le niveau supérieur de la hiérarchie des domaines.

2.5.2 Traitement des invocations au niveau système

Il y a deux manières de délivrer une requête à l'objet spécifié et de retourner un résultat éventuel :

- **L'envoi de messages**. Lorsqu'un client invoque un objet, le système construit un message dans lequel il insère les paramètres de l'invocation. Le message est alors envoyé à un serveur de processus ou a un port associé à l'objet invoqué qui traite alors la requête. Lorsque l'opération est terminée, un message contenant le résultat est retourné au client. Ce mécanisme est assez naturel, puisque c'est souvent de cette manière que fonctionne le mécanisme de communication entre les processeurs. Cependant, pour des invocations entre objets d'un même processeur, ce mécanisme est moins efficace que l'appel direct.

  Argus, *COOL*, et *Eden* supportent ce mécanisme.Il est à noter que l'invocation d'objets dans COOL est réalisée en utilisant les moyens de communications fournis par le noyau Chorus i.e. l'envoi de messages à travers des ports. Les comunications entre machines sont réalisées en envoyant les messages vers un *port local de substitution* qui est géré par le serveur réseau. C'est ce serveur qui est chargé d'acheminer les messages d'une machine à une autre d'une manière transparente.

- **L'appel direct**. Cela correspond à l'appel de procédure si l'invocation d'un objet est locale, et à l'appel de procédure à distance dans le cas contraire.

  Les SDO suivants utilisent ce mécanisme : *Amœba* (à travers des ports), *Clouds*, *Emerald* (dès qu'un processus appelle un objet, un *enregistrement d'invocation* est créé pour ce processus. Cet enregistrement est placé au sommet de la pile du processus si l'invocation est locale, dans le cas contraire il forme la base de la pile d'un nouveau processus sur la machine distante), *Guide* (invocation par *partage d'objets*. Lorsqu'une requête est destinée à un objet qui n'est pas local, le partage d'objet se fait uniquement par extension de l'exécution : l'activité appelante s'étend sur le site de l'objet invoqué), *PEACE* et *SOS* (l'invocation d'un service distant se fait en acquérant un mandataire approprié. Cela est réalisé de manière dynamique par migration d'un fragment dans le contexte de l'objet).

Le mécanisme d'appel direct semble le plus répandu. Il est sûrement lié au fait que l'appel de procédure, pour les communication locales, est beaucoup moins coûteux que l'envoi d'un message.

Une bonne partie des mécanismes de désignation des objets de ces SDO utilisent des identificateurs qui sont uniques dans tout le système et qui sont codés de manière hiérarchique (Amœba, COOL, PEACE) : un identificateur est un couple ("espace", identificateur local à l'espace et qui n'a un sens qu'à l'intérieur de cet espace). Ceci permet de gérer en parallèle la création des identificateurs (la construction d'un identificateur local à un domaine est indépendante des autres domaines), d'ajouter dynamiquement

des domaines, et d'augmenter les algorithmes de recherche des objets (puisqu'au lieu de comparer deux identificateurs dans leur totalité, il suffit d'abord de comparer leurs espaces. Si ceux-ci ne sont pas égaux, les identificateurs ne peuvent l'être).

Pour conserver la localisation d'un objet, la technique la plus répandue consiste à gérer un cache local et à diffuser une requête de localisation sur tout le réseau ou à s'adresser à un serveur de localisation en cas d'échec.

Certains SDO (COOL, PEACE) combinent même cette technique avec l'encodage dans l'identificateur de la localisation du site de création de l'objet et l'utilisation de liens de poursuite de l'information. afin de réduire les échecs de localisation.

De l'étude sur les interactions entre les objets, nous pensons que dans des systèmes parallèles où la mobilité des objets risque d'être importante, encoder la localisation d'un objet dans son identificateur et utiliser des liens de poursuite pour le retrouver, est une opération trop coûteuse car les pointeurs deviennent nombreux. De plus, en cas de panne d'un des processeurs sur lequel se trouve l'un des pointeurs, l'objet visé risque de ne jamais être retrouvé. De même, les mécanismes à base de diffusion (qui sont généralement utilisés lorsque une recherche dans un cache local a échoué) sont excessivement coûteux, et devront si possible être évités. C'est pourquoi nous avons décidé de supporter un mécanisme de désignation basé sur des serveurs de noms et des caches locaux. Enfin PARX étant basé sur l'envoi de messages, les invocations entre entités se feront principalement par envoi de messages. Nous verrons cependant, dans le **Chapitre 4**, que certaines communications utiliseront l'appel de procédure (cas des objets statiques).

2.6 Gestion des ressources

Un SDO, comme tout autre système distribué, doit fournir des mécanismes de gestion des ressources physiques de la machine (mémoire, mémoire auxiliaire, processeurs, etc.). Les SDO sont aussi responsables de la représentation des objets en mémoire principale et auxiliaire (disque), de la manière dont ils sont transférés entre les deux ressources, et de leur allocation sur les différents processeurs.

Les objets qui sont perdus lorsque la machine sur laquelle ils se trouvent tombe en panne, sont dit *volatils*. Les objets qui peuvent survivre à la panne d'une machine avec une haute probabilité sont dit *permanents*. Un objet permanent réside en mémoire auxiliaire ; cependant une ou plusieurs copies peuvent se trouver en mémoire. Cette version en mémoire secondaire est typiquement mise à jour lors de la procédure de validation.

Un objet permanent qui réside à la fois en mémoire primaire et secondaire est dit *actif*. S'il ne se trouve qu'en mémoire auxiliaire, il est dit *inactif*. Lorsqu'une invocation est faite sur un objet inactif, une copie volatile est créée et chargée en mémoire afin de rendre l'objet actif. De nouveaux processus sont aussi éventuellement créés si l'objet en mémoire en a besoin. Lorsque la procédure de validation s'est exécutée correctement,

l'objet est recopié en mémoire secondaire et peut être désactivé afin que le système récupère les ressources qui lui étaient associées. Ce chargement et déchargement automatique des objets est réalisé de manière transparente par le SDO afin de virtualiser l'utilisation de la mémoire et donner l'impression que les objets sont toujours disponibles.

Dans *Amœba*, un objet permanent est stocké en mémoire secondaire grâce à un mécanisme de reprise avec historique [Ko81].

Argus utilise un mécanisme de journal de bord avec validation [ko81] pour les objets permanents en mémoire secondaire. Une remise à jour périodique du journal est réalisée afin d'en réduire sa taille.

Clouds, *Emerald*, et *SOS* conservent les objets permanents en mémoire secondaire grâce à un pur mécanisme de points de reprise. Ce mécanisme est optimisé dans SOS, pour ne sauvegarder que les portions modifiées de l'objet. Clouds, quant à lui, supporte un système paginé : un objet est stocké en mémoire secondaire sous forme d'un ensemble de pages chargées en mémoire à la demande.

Dans *COOL* l'espace d'adressage d'un acteur (contexte) est constitué d'un certain nombre de régions non recouvrantes. Chaque région est généralement représentée en mémoire secondaire par des segments. Les objets sont conservés en mémoire secondaire (au sein de leur contexte) grâce à un pur mécanisme de points de reprise.

Eden conserve les objets permanents en mémoire secondaire grâce à un mécanisme de reprise avec historique géré par un serveur de fichiers. Chaque objet est géré par un *gestionnaire de version* qui contrôle les accès à la version la plus récemment validée et à toutes les versions non validées de l'objet. Les objets sont accédés grâce au gestionnaire de version et peuvent être ouverts, fermés ou validés. lorsqu'une action modifie un objet, l'objet est ouvert et une copie de sa version courante est copiée en mémoire. Lorsque l'action se termine, l'objet est fermé et une nouvelle version non validée de l'objet est créée. Lorsque toutes les actions qui affectent un objet sont terminées et validées, la plus récente des versions non validées devient la version courante.

Dans *Guide*, les objets permanents sont stockés dans une mémoire secondaire multi-site à accès transparent (appelée *mémoire permanente d'objets*), et sont chargés à la demande dans une *mémoire virtuelle d'objets* pour leur exécution.

2.6.1 Représentation des objets en mémoire principale

La représentation d'un objet en mémoire dépend de la manière dont sera assurée la synchronisation (influence sur le nombre de versions d'un objet maintenues en mémoire) et dont la gestion des actions sera faite (influence sur la représentation de chaque version).

Sur des architectures à mémoire distribuée, lorsqu'on emploie un mécanisme de synchronisation pessimiste, une seule version de chaque objet est maintenue en mémoire.

Dans ce cas, toute action qui invoque cet objet, modifie la même version volatile. Lorsqu'on utilise le mécanisme de synchronisation optimiste, de multiples versions de chaque objet sont créées et maintenues en mémoire. Ceci permet à de multiple actions d'invoquer le même objet simultanément tout en garantissant qu'elle n'interféreront pas entre elles.

Si on n'utilise pas le mécanisme transactionnel emboîté [Mo85], une version en mémoire est une copie exacte de l'objet correspondant. Si une action échoue, la version volatile est simplement éliminée. Dans le cas contraire, il faut en plus utiliser un mécanisme qui enregistre toutes les modifications, afin de pouvoir défaire les changements effectuées par une sous-action qui a échouée.

Les SDO suivants gèrent de multiples versions d'un objet : *Amœba*, *Argus*, *Eden* . Les SDO suivants gèrent plusieurs copies d'un objet : *PEACE*, *SOS* (fragments élémentaires). Quant à, *Clouds*, *COOL*, *Emerald*, et *Guide*, ils gèrent les objets présents en mémoire en un exemplaire.

2.6.2 Les processeurs

Le principal objectif de la gestion des processeurs est de maximimer le rendement du système en minimisant le temps d'attente des objets pour recevoir les services du processeur. Le placement des objets sur les différents processeurs est rendu difficile par deux objectifs contradictoires : premièrement les objets devraient être placés sur des processeurs différents et peu chargés afin qu'ils puissent s'exécuter en parallèle, et deuxièmement, les objets qui interagissent fréquemment devraient être placés sur le même processeur ou sur de proches voisins, afin de réduire les coûts de communication. Pour une performance optimale, les objets d'un programme devraient être assignés à un groupe de processeurs, étroitement liés et peu chargés.

Ainsi par exemple, *Amœba* maintient un *groupe de processeurs* dans lequel un client peut dynamiquement réclamer un certain nombre de processeurs disponibles, ces processeurs ne pouvant être choisis. Lorsqu'un processeur n'est plus nécessaire, il est rendu au *groupe*.

2.6.2.1 Placement des objets

Dès qu'un objet est créé ou qu'un objet inactif est activé il doit être placé sur un processeur. Généralement un nouvel objet est placé sur le processeur où il a été créé ; cependant le système peut aussi autoriser une création sur un processeur distant. Un objet actif peut être placé sur n'importe quel processeur de même type que celui sur lequel il a été créé, exception faite des *objets immobiles* (objets dont la localisation est directement codée à l'intérieur de leur identificateur) qui sont toujours placés sur le même processeur.

Le placement des objets peut être soit *explicite* (l'utilisateur est responsable de la désignation du processeur sur lequel l'objet doit être placé) soit *implicite* (c'est le système qui décide du placement des objets en fonction d'un certain nombre de critères).

De nombreux SDO autorisent le placement explicite d'objets ; c'est le cas pour *Argus*, *Clouds*, *COOL*, et *Guide* (il est aussi possible de déclarer un objet "*inamovible*", notion similaire à la notion d'objet immobile précédemment introduite).

2.6.2.2 Mobilité des objets

Un mécanisme de *migration des objets* permet de déplacer des objets d'un processeur à un autre, à n'importe quel moment, parfois même lorsqu'ils sont en cours de traitement d'une invocation. L'avantage de la migration réside dans le fait qu'elle augmente les performances (par exemple en diminuant la charge des processeurs ou en réduisant les coûts de communication entre objets) et la disponibilité (par exemple en déplaçant des objets d'une machine qu'on va arrêter) du système. Pour une liste exhaustive d'algorithmes de migration, le lecteur pourra consulter [El92].

La migration est supportée dans : *COOL*, *Eden*, *Emerald* (pour simplifier cette tâche, du code relogeable est généré et des *gabarits* qui décrivent la structure interne des objets sont créés. Il est possible de rendre immobile un objet en le "*fixant*" sur un processeur donné. Il est aussi possible *d'attacher* un objet global [OG1] à un autre objet global [OG2], de telle sorte que la migration de OG2 entraîne automatiquement la migration de OG1), *Guide* (pour être plus exacte, la seule migration possible est la *migration verticale* des objets entre une unité de stockage et un site d'exécution), *PEACE* et *SOS* (Les objets peuvent migrer d'un contexte à un autre. La migration d'un objet élémentaire peut entraîner la migration d'autres objets élémentaires, dits "*prérequis*". Il est possible d'empêcher la migration d'un objet en le déclarant "*non-mobile*").

Aucune tendance forte ne se dégage pour la représentation des objets tant en mémoire primaire que secondaire. Les techniques utilisées pour la gestion des objets permanents proviennent généralement de l'univers des base de données.

En revanche, la plupart des SDO fournissent au moins un algorithme de placement des objets à leur création, ceci afin d'équilibrer au mieux la charge du réseau de machines. Certains SDO (COOL, Eden, Emerald, PEACE, SOS) offrent même un algorithme de migration des objets, pour améliorer le placement initial, et permettre à l'utilisateur un contrôle plus fin de l'exécution de son programme parallèle. La migration des objets a entraîné les concepteurs de SDO à introduire la notion d'objet **fixé** à un site qui permet d'empêcher sa migration, et la notion **d'attache** entre objets, pour que la migration d'un objet entraîne automatiquement la migration des objets qui lui sont attachés.

De l'étude sur la gestion des ressources, nous pensons qu'un SDO qui ne permet pas aux objets d'être placés sur un site particulier à leur création, ou migrés durant leur durée de vie, ne sera pas très performant, car il ne permet pas d'équilibrer au mieux la charge du réseau, et ne laisse pas à l'utilisateur la possibilité de contrôler finement l'exécution de son application parallèle. Nous supporterons donc ces mécanismes. Dans

notre implémentation, les entités n'auront pas leur localisation encodée dans leur identificateur, ceci afin d'éviter les liens de poursuite, que nous avons jugé trop pénalisants. Enfin, les processeurs seront gérés grâce à la notion de **cluster** (groupe de processeurs connectés entre eux, fournissant un domaine de communication et de protection et dont le nombre peut évoluer au cours du temps) de PARX.

2.7 tableau récapitulatif des SDO cités

Avant de résumer, dans un tableau synthétique, les aspects qui ont été traités dans ce chapitre (structure des objets utilisés, gestion des objets, interactions entre ces objets, et gestion des ressources), nous nous permettons de décrire brièvement les motivations et éventuellement les architectures visées par ces systèmes. Le lecteur pourra se référer à [BN89] pour une description plus détaillée de certains de ces SDO.

L'objectif initial d'**Amœba** [TM80] est de produire un système réparti intégré pour des machines hétérogènes dans un réseau local.

Argus [Li88] est un langage de programmation et un système développé pour supporter l'implantation et l'exécution de programmes distribués.

Clouds [Da90] est un SDO constitué d'un noyau minimal appelé Ra (qui fournit des fonctionnalités de base comme la gestion mémoire virtuelle, les processus légers, l'ordonnancement de bas niveau, etc.), et d'un ensemble d'objets système qui fournissent les services système.

COOL ("Chorus Object-Oriented Layer") [HMA90] est une extension construite au dessus du micro-kernel Chorus [Chorus89], [AGHR89], et qui offre un support pour les systèmes distribués orientés-objets.

Eden [ABLN85] est un système qui fournit à la fois un noyau distribué sur un ensemble de VAX tournant sous UNIX et connectés par Ethernet, et un langage de programmation (appelé EPL).

L'objectif d'**Emerald** [BHLC87], [JLHB88] est de démontrer la faisabilité de l'utilisation d'un modèle sémantique simple pour à la fois la programmation d'applications séquentielles sur un monoprocesseur, et la programmation d'applications parallèles sur des réseaux de processeurs. Il est réalisé au dessus d'UNIX sur un réseau local de stations de travail MicroVax.

La motivation du projet **Guide** [De90], [KMNR90] est l'exploration des problèmes techniques et scientifiques relatifs au support d'application distribuées sur un réseau de machines éventuellement hétérogènes. En particulier, il doit masquer la plupart des problèmes relatifs à la distribution en offrant une transparence d'accès et d'utilisation des ressources. Guide fonctionne au dessus d'UNIX sur un réseau local de stations de travail.

L'objectif principal de **PEACE** [Sc90], [Sc91], [Sc92] est de fournir un environnement de communication et d'exécution de processus ("**P**rocess **E**xecution **A**nd **C**ommunication **E**nvironment") pour des applications distribuées extensibles.

SOS [SGHM89], [Ha89] est un SDO basé sur le principe du *mandataire* : l'interface à un service, réalisé par un ensemble d'objets coopérants, est un objet de communication unique, un "mandataire" pour ce service. SOS est réalisé en C++ au dessus du système UNIX (SunOS).

		Amœba	**Argus**	**Clouds**	**COOL**	**Eden**	**Emerald**	**Guide**	**PEACE**	**SOS**
Structure des objets	*Granularité* (***G**ros, **M**oyen, **F**in)*	G	GM	G	M	G	tous	MF	tous	M
	Comportement (***P**assif, **A**ctif*[6]*)*	A/s	A/d	P	P	A/s	P A/s	P	P A/s	P
Gestion des objets	*Synchronisation* (***O**ptimiste, **P**essimiste)*	OP	P	P[7]	P	P	P	P[5]	P	P
	Protection (***C**apacité, **P**rocédure)*	C	—	C	C	C	—	—	C	C
Gestion des interactions	*Localisation* (***E**ncodé, **S**erveur, **C**ache, **D**iffusion, **L**ien de poursuite)*	E CD	E	CD	E LD	CD	L CD	E	EL CS	EL CD
	Invocation (***M**essage, **D**irect)*	D	M	D	M	M	D	D	D	D
Gestion des ressources	*Représentation* (***P**ermanent, **V**olatile / **U**nique, **M**ultiple)*	P M	P M	P U	P U	P M	P U	P U	V M	P M
	Processeur (***P**lacement, **M**igration)*	—	P	P	PM	M	M	P	M	M

6 **s**tatique ou **d**ynamique

7 Personnalisable

II. ParObj : un noyau de système parallèle à objets

Chapitre 3 : Architecture du noyau PARX

3.1 Motivations

PAROS (et son noyau PARX [LM88]) ont été développé dans le cadre du projet ESPRIT Supernode II (P2528) [SuII91], et dont l'objectif est la conception d'un système d'exploitation efficace autonome et non dédié, pour machines parallèles sans mémoire commune. Le but est d'atteindre de hautes performances pour l'exécution d'applications parallèles, grâce à la construction correcte de mécanismes nouveaux pour la gestion du parallélisme, de la communication et des ressources dans des machines massivement parallèles hétérogènes et extensibles.

Pour atteindre cet objectif, il a fallu s'intéresser aux aspects suivants :

1. Définir les supports adéquats et efficaces pour la gestion des différents grains de parallélisme et de communication dans les applications parallèles [La91], [Go91],

2. Fournir des niveaux variés de construction de sous-systèmes pour des systèmes d'exploitation standards existants (X/OPEN, POSIX) ou des environnements de programmations (PCTE[8]), par la conception du noyau PARX comme un ensemble hiérarchique de machines virtuelles offrant un support d'exécution correcte pour divers modèles de programmation.

3. Contribuer à l'évolution vers un standard pour un système d'exploitation parallèle sur les plus récentes des architectures parallèles sans mémoire commune (à base de transputeurs ou d'autres processeurs) développées en Europe (par exemple les machines Supernode actuelles ou les futures plates-formes à base de T9000).

Or, les systèmes d'exploitations couramment utilisés (par exemple Unix) n'offrent qu'un usage limité du pseudo-parallélisme et des communications : le paradigme de programmation normalement supporté par ces systèmes est basé sur des processus séquentiels communiquant faiblement entre eux à travers des objets de communications lourds à mettre en œuvre (*sockets* par exemple).

8 **P**ortable **C**ommon **T**ool **E**nvironment

Avec l'arrivée des systèmes distribués (Chorus, Mach ou Amœba par exemple), un nouveau modèle d'exécution apparaît, qui permet l'exploitation du pseudo-parallélisme et du parallélisme vrai, grâce à l'utilisation d'objets de communication et de moyens de gestion de la concurrence à grain beaucoup plus fin (que ceux utilisés dans les systèmes traditionnels). La structure d'un processus est alors éclatée en deux entités : la première est apparentée à la gestion de l'environnement (contexte d'exécution et ressources allouées), la seconde est apparentée aux flots de contrôle, qui regroupent différentes activées "légères" s'exécutant dans le même contexte (Acteur/Threads dans Chorus, Tâche/Threads dans Mach, Processus/Threads dans Amœba).

Cependant, ces systèmes distribués ne résolvent pas tous les paradigmes imposés par les systèmes parallèles. En particulier, Ils n'offrent pas la vision d'une machine parallèle aux utilisateurs. Or cette vision est très utile à un ensemble important d'applications existantes qui requièrent un contrôle direct des ressources de la machine.

Un autre inconvénient qui apparaît dans les systèmes distribués, est le manque d'un mécanisme approprié de contrôle des applications parallèles (placement des processus sur les processeurs de la machine, lancement et destruction de processus, synchronisation de processus, routage automatique des messages, etc.). Ces fonctionnalités sont souvent complexes et nécessitent d'être réalisées d'une manière efficace afin de prendre en compte les performances de la machine.

A cet égard, la parallélisation d'un application devrait s'accompagner de la parallélisation de son environnement. Par exemple, un gestionnaire de fichier peut devenir un sérieux goulot d'étranglement pour des applications parallèles, s'il ne supporte pas l'accès en parallèle à plusieurs fichiers.

Un legs important de l'expérience acquise dans les systèmes distribués est la notion de micro noyau. Dans un système distribué, le gros des services système sont exécutés en mode utilisateur. Seul, un ensemble réduit de services s'exécutent en mode superviseur (le micro noyau). Cette organisation permet de supporter une structure système tolérante aux pannes, extensible et orientée application.

Cependant, bien que les micro noyaux soient bien adaptés pour les systèmes distribués, ils apparaissent "trop lourd" pour le support d'architectures distribuées massivement parallèles : ils introduisent en effet un surcoût qui ne leur permet pas de rester proches des performances de la machine nue.

PAROS a été pensé dans le but de faire face aux problèmes soulevés. Afin de satisfaire les applications qui nécessitent des performances élevées, et de faciliter la programmation sur machines parallèles, une approche basée sur un *noyau générique parallèle* a été choisie. Le noyau PARX permet la construction correcte d'un ensemble de **machines virtuelles** (ou niveau d'abstraction) grâce à un *mécanisme générique de construction de protocoles*. Ainsi chaque utilisateur à la vision d'une machine parallèle virtuelle la mieux adaptée aux besoins spécifiques de ses applications.

3.2 Architecture générale de PAROS

PAROS permet de considérer un ordinateur parallèle comme une machine unique. A l'intérieur de cette machine, les processeurs sont juste assimilés à des ressources supplémentaires qu'il faut gérer. Ainsi, PAROS offre une vision très uniforme de la machine dans laquelle les ressources peuvent être utilisées par n'importe quel programme en cours d'exécution (ou partie d'un programme parallèle), indépendamment de la localisation physique de ce programme ou des ressources auxquelles il accède. Ceci inclut par exemple, la création de processus et l'allocation de portion de mémoire à distance, l'envoi et la réception de messages, etc.

Afin d'être indépendant de la disponibilité des ressources physiques de la machine, PAROS établit une séparation franche entre les concepts de processeurs logique et physique, de la même manière que les espaces d'adressage logique et physique sont différentiés dans les architectures traditionnelles. Cette séparation ajoute une grande flexibilité dans la gestion des ressources. Ainsi, le noyau peut établir une correspondance entre les processeurs logiques (sur lesquels s'exécute une application), et les processeurs physiques disponibles d'une machine, indépendamment de sa structure physique, ce qui permet le portage immédiat d'applications tournant sous PAROS mais s'exécutant sur des machines (parallèles ou non) différentes. Toutefois, il est aussi possible d'avoir un niveau d'interface de machine virtuelle qui permet aux utilisateurs préoccupés par le placement logique/physique d'avoir un contrôle sur ce placement.

La figure 3.1 décrit l'architecture générale de PAROS. Le système a été structuré en un noyau "allégé" et un ensemble d'environnement de sous-systèmes construit au dessus. Le noyau PARX fournit un ensemble réduit d'abstractions de base simples et bien définies qui peuvent être utilisées par les applications et les développeurs de sous-systèmes.

PARX est découpé en différents niveaux de machines virtuelles qui offrent des interfaces assurant la compatibilité des programmes utilisateurs avec divers modèles ou environnements de programmation, tous supportés comme des sous-systèmes (PCTE, X/OPEN par exemple). Ces sous-systèmes peuvent coexister au dessus d'un ou plusieurs niveaux de machine virtuelle offert par PARX. Ainsi, chaque sous-système offre une vue particulière de la machine. Cela pourra être une machine Unix standard, ou bien une machine base de données ou encore une machine numérique.

Le niveau le plus bas représente la machine virtuelle d'extension du matériel (HEM[9]) qui peut être aussi bien un ordinateur parallèle à mémoire distribuée (DMPM[10]) qu'à mémoire commune (SMPM[11]).Il ne présuppose aucun modèle de programmation pour l'exploitation des fonctionnalités de la machine. Il offre la vue d'une machine abstraite unique, constituée d'un ensemble de processeurs entièrement connectés virtuellement,

9 **H**ardware **E**xtension **M**achine
10 **D**istributed **M**emory **P**arallel **M**achine
11 **S**hared **M**emory **P**arallel **M**achine

de mécanismes de communication, de protocoles de base pour l'échange de messages et de facilités pour l'activation de processus entre n'importe quelle paire de processeurs. En bref, il encapsule les fonctionnalités "machines dépendantes" des processeurs.

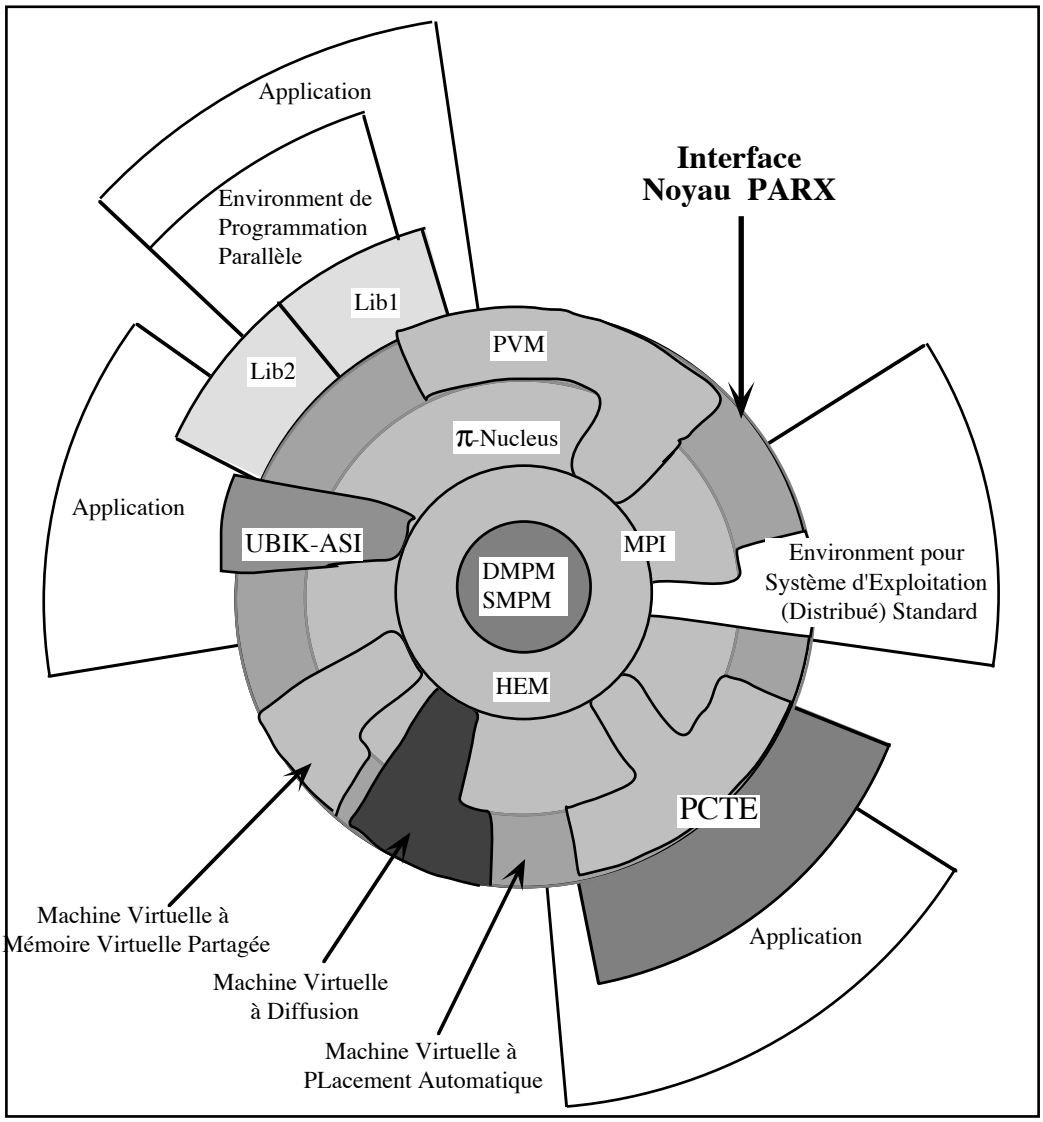


Figure 3.1. Architecture générale de PAROS

Comme les micro-noyaux des systèmes distribués, le micro-noyau de PAROS (π-nucléus) n'offrent pas les fonctionnalités traditionnelles de haut niveau des systèmes d'exploitation (gestionnaire de fichiers, accès réseau, etc.). Il fournit cependant le support pour les réaliser en tant que sous-systèmes au dessus du micro-noyau. Le π-nucléus met en œuvre les politiques d'allocation des ressources de base (processeurs, mémoire, etc.), et garde un contrôle sur les entités de base du système (processus et objets de communication). Des politiques d'allocation spécifiques peuvent aussi être implantées à ce niveau, pour par exemple supporter l'ordonnancement d'applications temps réel.

Le π-nucléus offre un mécanisme générique pour la construction des protocoles utilisés par les diverses machines virtuelles (diffusion, rendez-vous réparti, transfert de données

distribuées, protocole clients-serveur, etc.). Le modèle de processus supporte directement le concept de programme parallèle constitué de plusieurs espaces d'adressages, ainsi que le concept de flots d'instructions très légers qui s'exécutent à l'intérieur de ces espaces d'adressage.

Dans les niveaux supérieurs de PARX, les autres services supportés par les systèmes d'exploitation peuvent être ajoutés. C'est dans ces couches que différentes interfaces standards peuvent être construites sous la forme de machine virtuelle de PARX. Machines virtuelles à placement automatique, à mémoire virtuelle partagée et à diffusion sont quelques exemples d'interfaces développées.

3.3 Le noyau PARX

Dans ce paragraphe, nous donnons une brève description des concepts et de base et des fonctionnalités de PARX.

3.3.1 Modèle de processus

L'architecture générale de PARX est basée sur modèle original à trois niveau de processus. Il offre les concepts appropriés et le support correct pour des applications qui exhibent différents grains de parallélisme. Le modèle de processus comporte trois abstractions : le *thread*, la *task* (ou tâche) et la *P_task* (ou tâche parallèle[12]).

Un **thread** est un flot de contrôle séquentiel à l'intérieur d'une tâche. Un thread peut être vu comme un processus "très léger". Les threads ne permettent pas l'expression du parallélisme. L'utilisation de plusieurs threads à l'intérieur d'un tâche est un moyen efficace pour supporter certains constructeurs de langages parallèles (le PAR d'OCCAM par exemple), pour réaliser les programmes multi-threads (objets actifs par exemple), ainsi que pour facilement implanter des communications asynchrones au dessus des mécanismes de communication synchrones fournis par le noyau PARX. Un thread est une unité de pseudo-parallélisme et d'ordonnancement.

Une **tâche** est contexte d'exécution dans lequel peuvent évoluer plusieurs flots de contrôle concurrents (les threads). Les threads à l'intérieurs d'une tâche communiquent par mémoire partagée. Une tâche s'exécute sur un processeur virtuel donné à l'intérieur d'une P_task. La tâche est à la fois un espace d'adressage logique et une unité de parallélisme et d'ordonnancement.

Une **P_task** représente un programme parallèle en cours d'exécution sur une machine virtuelle. Une Ptâche est une entité administrative qui gère les ressources des processus parallèles d'un même programme. Elle définit un domaine de communication et de protection. Elle est représentée par un ensemble de tâches et de protocoles de communi-

12 Que nous appellerons souvent **Ptâche** par commodité d'écriture.

cation et de synchronisation entre ces tâches. Le rôle de la Ptâche est de contrôler le lancement des tâches, leur exécution en parallèle, les protocoles d'échange entre celles-ci ou entre leurs threads, et leur terminaison correcte ; la Ptâche assure ainsi la correction sémantique d'un programme parallèle en exécution et supporté par PARX.

L'ordonnancement à l'intérieur d'une Ptâche est réalisé à deux niveaux : pour les threads et pour les tâches. Ceci permet par exemple l'utilisation, au niveau thread, s'ils sont disponibles, de dispositifs comme les ordonnanceurs matériels. L'ordonnanceur permettra aussi aux sous-systèmes de fournir leur propre politique d'ordonnancement pour accroître les fonctionnalités du modèle par défaut.

Ce modèle de processus, résumé sur la figure 3.2, est suffisamment général pour permettre aux programmeurs (et aux applications déjà écrites) de s'interfacer avec les modèles de processus d'autres systèmes d'exploitation (acteur/threads de Chorus, tâche/threads de Mach, processus/threads d'Amœba, etc.)

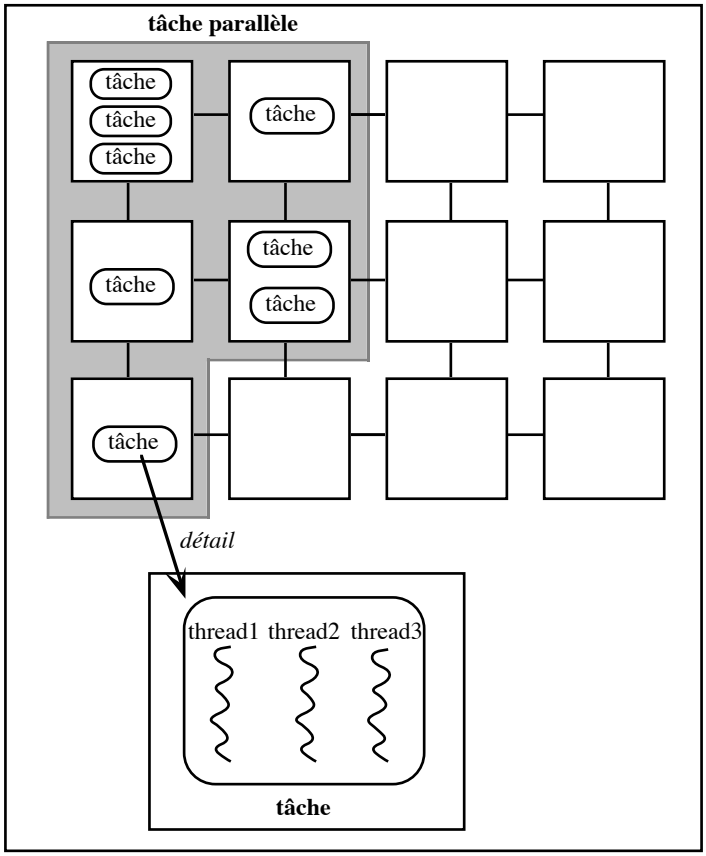


Figure 3.2. Ptâche, tâches et threads dans PARX

3.3.2 Modèle de communication

Le modèle de communication entre processus a été clairement conçu pour être cohérent avec le modèle de processus. Ce modèle est, à la base, constitué d'un ensemble de processus communicants qui collaborent pour établir des protocoles de communication sur une architecture dont la communication se fait par échange de messages. Les algorithmes de routage utilisés sont corrects (sans interblocage) et prévus pour des machines extensibles et de topologie quelconque. A l'intérieur d'une Ptâche, les protocoles ne sont pas redondants et n'ont pas d'effets de bord sur les autres Ptâches.

Au dessus du niveau d'acheminement des messages, des protocoles plus complexes sont construits de façon générique [BFFG89]. Les développeurs de sous-systèmes peuvent utiliser cette interface de bas niveau, pour bâtir des protocoles plus sophistiqués destinés à leur propre usage, ou encore pour construire des bibliothèques utilisateurs.

La figure 3.3. présente quelques protocoles déjà réalisés : le rendez-vous entre processus distant (rendez-vous CSP), le transfert de données distribuées (**D**istributed **D**irect **M**emory **A**ccess), le protocole clients-serveur (**A**synchronous **S**erver **P**rotocol) et la diffusion.

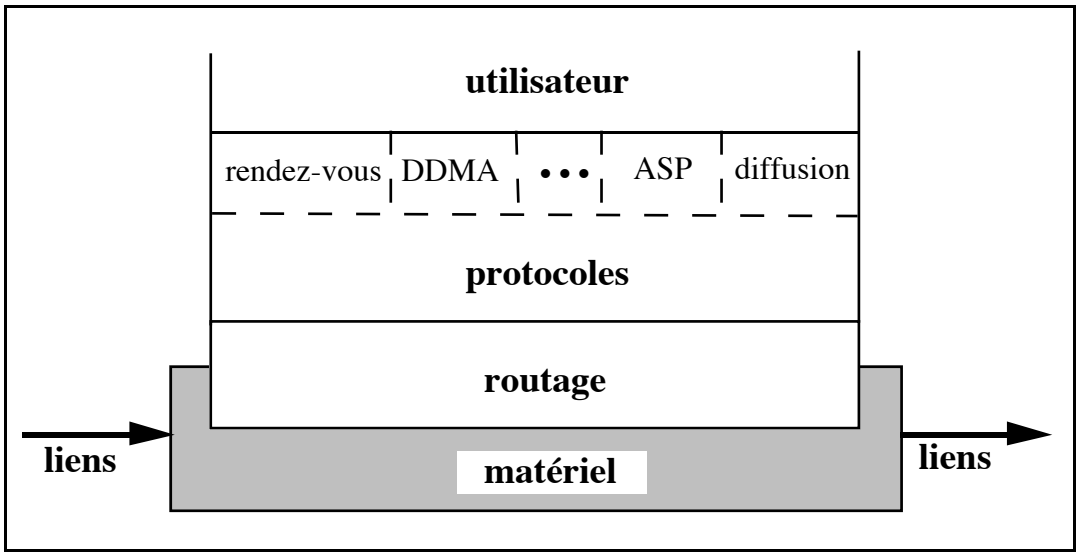


Figure 3.3. Découpage en niveaux du modèle de communication

Deux objets de communication de base ont été choisis pour supporter un ensemble relativement large de machines virtuelles : les **canaux** et les **ports**. Ils sont tous les deux *synchrones*. Les ports sont un mécanisme global et flexible de communication orienté système, plusieurs vers un et protégé. Ce mécanisme est utilisé pour accéder aux sous-systèmes et aux serveurs. Les canaux sont un mécanisme rapide de communication, un vers un, entre tâches ou threads à l'intérieur d'une même Ptâche. Notons que les threads appartenant à une même tâche peuvent communiquer aussi bien par un canal de mémoire partagée que sur un canal de transfert de données. Les canaux sont locaux à l'intérieur d'une Ptâche et ne peuvent être utilisés comme moyen de

communication entre Ptâches ; le seul moyen de communication entre Ptâches est le port (Cf. figure 3.4).

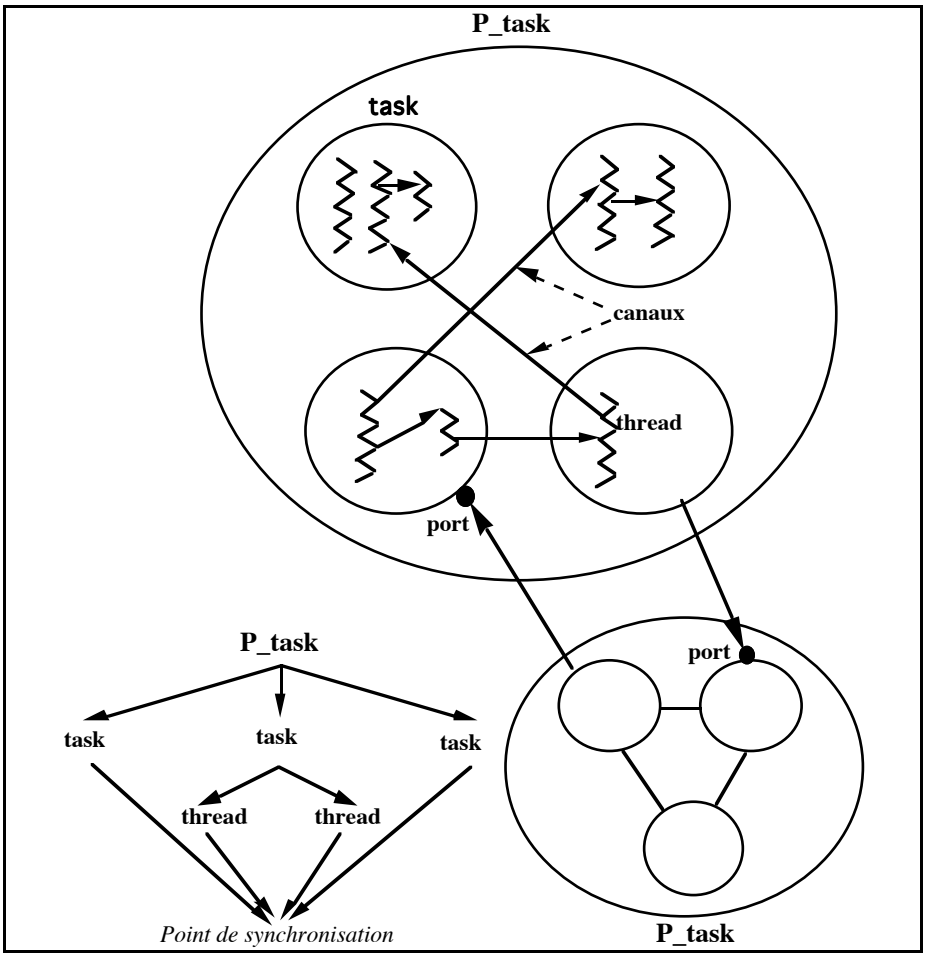


Figure 3.4. Objets de communication entre Ptâches, tâches et threads

3.3.3 Gestion des processeurs

Deux types de regroupement de processeurs sont à distinguer : un *bouquet* de processeurs physiques nus (**bunch**) ou une *grappe* d'espaces d'adressage disjoints (**cluster**) localisés sur un ensemble connexe de processeurs du réseau (statiquement ou dynamiquement configurable), Cf. figure 3.5.

3.3.3.1 Bunch

Un bunch est un ensemble de processeurs et de connexions entre ces processeurs qui fournit un support "machine nue". On peut ainsi amener PAROS à être compatible avec les actuels environnements parallèles (C 3L, C INMOS dans le cas du transputeur).

Une application parallèle existante sera directement portable sur un bunch sans avoir à la réécrire. De plus, certains utilisateurs préféreront le bunch afin d'exécuter des pro-

grammes parallèles avec un maximum d'efficacité (calcul scientifique par exemple) sans payer le surcoût nécessairement introduit par les fonctionnalités offertes par le noyau d'un système.

3.3.3.2 Cluster

C'est la machine virtuelle offerte pour l'exécution des programmes parallèles , et qui est constituée d'un ensemble de processeurs avec une configuration donnée et un support noyau de système complet (gestion des ressources, protocoles spécifiques de communication et de synchronisation, domaine de protection, etc.).

La fonction principale d'un cluster est d'offrir un support physique de telle sorte que seule la structure logique d'une tâche parallèle ait besoin d'être spécifiée. Le programme correspondant sera placé par le système ou par les utilisateurs sur les ressources physiques disponibles au moment du chargement.

Un cluster est donc une entité dynamique dans laquelle les processeurs physiques sont alloués et libérés durant l'exécution de la Ptâche. Cette propriété est utile pour améliorer l'utilisation des ressources inoccupées, pour supporter l'implantation de mécanisme de tolérances aux pannes, etc.

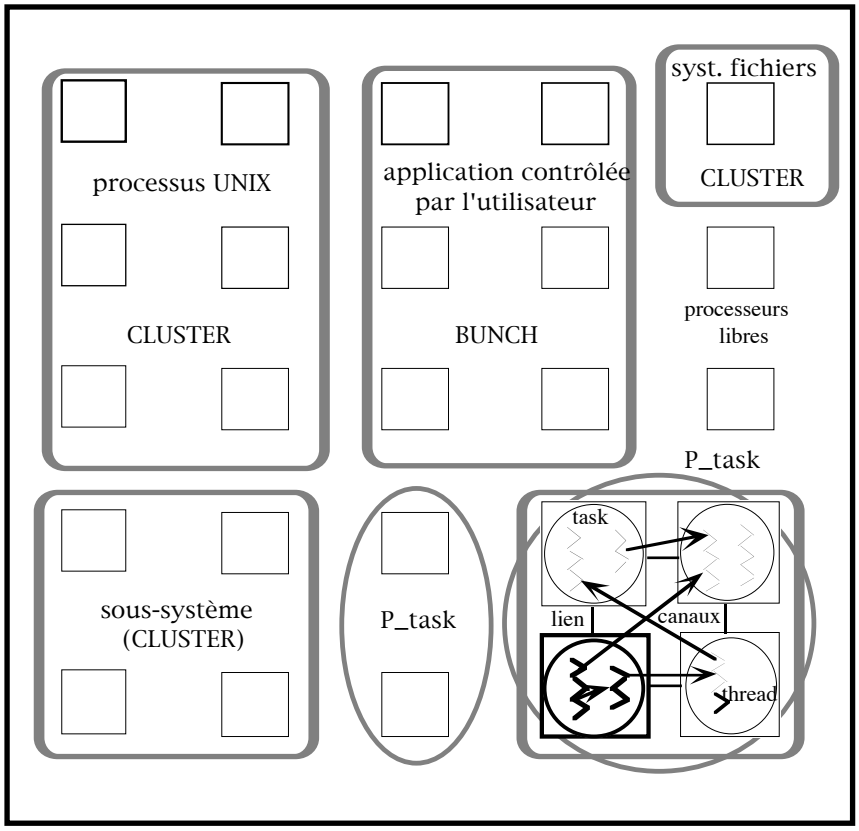


Figure 3.5 Bunchs et clusters dans PARX

3.3.4 Gestion mémoire

La mémoire est organisée en *segments*. Un segment est un ensemble d'adresse mémoire contiguës sur un même processeur. Un segment a un identificateur local à une Ptâche. Les segments peuvent être placés dans l'espace d'adressage des tâches avec une sémantique de *copie*. A chaque instant, un segment appartient à l'espace d'adressage d'une tâche et d'une seule.

L'interface pour la gestion des segments permet le chargement de code sur les processeurs. Elle est disponible aux développeurs de sous-systèmes.

Chapitre 4 : Les objets de ParObj

4.1 Un modèle hybride

Dans ce paragraphe, nous allons montrer ce qu'un modèle à objet peut apporter au modèle de processus à trois niveaux de PARX.

Comme nous l'avons vu dans le **Chapitre 2**, la programmation objet présente un certain nombre d'avantages (structuration, abstraction, transparence, persistance, encapsulation des données, mécanisme d'héritage, etc) par rapport à la programmation classique.

Cependant la programmation objet, comme la programmation parallèle, imposent à l'utilisateur des contraintes pas toujours désirées. Ainsi, par exemple, l'obligation d'utiliser des primitives d'envoi et de réception de message pour communiquer entre deux entités au lieu de passer par des variables globales par exemple, peut surprendre, voir même dérouter, tant cette manière d'écrire est peu "naturelle"; de même, penser son programme uniquement en terme d'objets et d'interactions entre ceux-ci, nécessite parfois un gros effort d'analyse.

Or il n'est pas sûr que le développeur occasionnel (ou même chevronné) soit capable ou simplement veuille se plonger dans les arcanes de la magie "objet" ou parallèle pour écrire ses applications. Ce n'est pas en effet, parce que nous disposons d'une machine parallèle que nous devons forcément complètement changer notre manière de programmer. C'est pourquoi, un système d'exploitation parallèle devrait être capable de fournir un certain nombre de fonctionnalités (comme par exemple une gestion mémoire virtuelle distribuée) afin que l'écriture d'applications parallèles ne présente pas plus de difficultés que l'écriture d'une application sur une architecture traditionnelle.

Mais on peut aussi exiger d'une machine parallèle encore plus d'efficacité dans sa manière de gérer un programme parallèle. Comme un système d'exploitation ne peut généralement pas satisfaire tous les désirs du programmeur, il est donc nécessaire de disposer d'un langage efficace et d'un certain nombre de constructeurs de ce langage (ou d'outils spécifiques) qui permettent une programmation parallèle optimale en renseignant le système et en l'aidant dans sa tâche (en donnant par exemple la listes des enti-

tés qui constituent un programme parallèle et en spécifiant comment celles-ci doivent être placées sur les différents processeurs).

En résumé, il apparaît qu'une bonne gestion du parallélisme doit se faire autant au niveau du langage (utilisation d'objets, opérateurs de parallélisation, etc.) qu'au niveau du système d'exploitation (allocation d'objets aux processeurs, serveurs d'objets et de noms pour résoudre les problèmes de désignation, etc.).

4.1.1 Un exemple

Supposons qu'un utilisateur habitué à travailler sous Unix, veuille maintenant porter une application spécifique sur une machine parallèle. Dans un premier temps, il voudra simplement vérifier qu'elle s'exécute correctement, sans chercher à la paralléliser. Dans ce cas, à l'exécution, son application sera une Ptâche s'exécutant sur un seul processeur et ne contenant qu'une seule tâche (l'ex-processus Unix). Le modèle de processus de PARX, bien qu'utilisé d'une manière dégénérée, remplit parfaitement son rôle.

Supposons maintenant, que ce même utilisateur décide de paralléliser son application, afin d'augmenter les performances de son programme. Sa Ptâche contiendra maintenant plusieurs tâches qui s'exécuteront sur plusieurs processeurs (soit parce que l'utilisateur l'aura expressément demandé, soit plus probablement parce qu'un outil ad-hoc aura détecté un parallélisme implicite). Dans ce deuxième exemple, on constate que le modèle de processus à trois niveaux reste adapté.

Supposons enfin que notre utilisateur devenu un programmeur parallèle chevronné, et disposant d'une machine à 16 processeurs, décide de se lancer dans l'écriture d'un éditeur de texte parallèle performant. Après quelques mois d'efforts, l'éditeur est enfin réalisé. Il a juste un petit défaut : pour être vraiment efficace, il a besoin d'au moins 7 processeurs. Supposons qu'un autre de ses collègues utilise aussi l'éditeur (Cf. figure 4.1). Dans ce cas, deux utilisateurs vont monopoliser 88% (14/16) des processeurs de la machine. Inutile de vous précisier que les autres utilisateurs ne sont plus tout à fait attirés par cet éditeur…

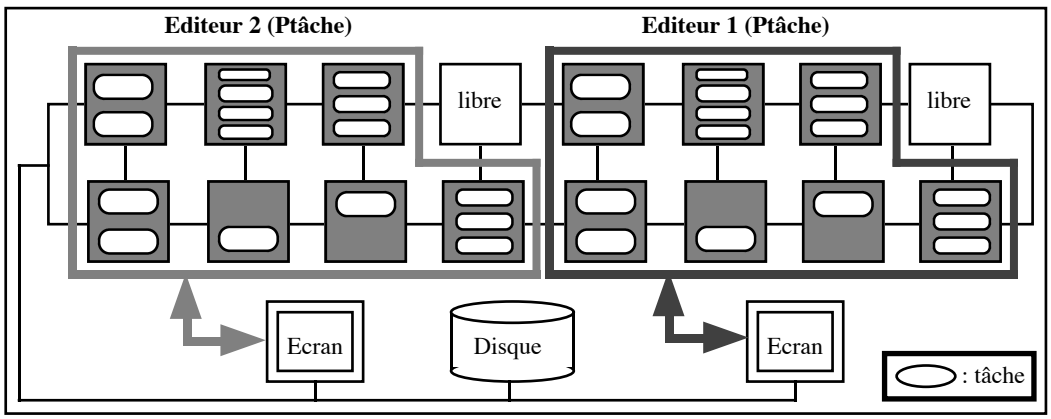


Figure 4.1. Deux éditeurs occupant 88% des processeurs

Notre programmeur pourra alors se dire : “si nous pouvions partager le code de l’éditeur, et travailler sur des données différentes, alors nous devrions gagner de la place. Et si d’autre part, les données (ici le document) pouvaient être suffisamment autonomes, afin par exemple de rafraîchir automatiquement l’écran dans lequel se trouve le document, alors l’éditeur fonctionnerait encore plus vite”. Malheureusement avec ce seul modèle pour la gestion des processus, il peut difficilement s’en sortir.

Supposons maintenant que le langage autorise la définition d’objets actifs.

Alors, notre développeur peut stocker le document sous forme d’un objet actif, avec des méthodes de manipulation de texte comme interface (`insérer(…)`, `détruire(…)`, `rechercher(…)`, etc) et des processus de mise-à-jour automatique de l’écran, lorsque le contenu du document est modifié. Il peut aussi utiliser d’autres objets pour gérer la table des matières, le plan du document, le résumé, etc. En revanche, il peut tout-à-fait conserver le modèle de processus à trois niveaux pour la partie purement gestion de l’éditeur (saisie des caractères, interprétation des commandes, …) qui pourra alors travailler plus efficacement puisqu’elle n’aura plus par exemple, à se préoccuper des problèmes de rafraîchissement d’écran.

Ainsi lorsqu’un deuxième utilisateur lance cette nouvelle version de l’éditeur, le système ne lui alloue plus, par exemple, que 3 processeurs, car ceux-ci seront suffisants pour stocker les objets contenant les informations relatives à son ou ses documents. Du coup, le taux d’occupation de la machine tombe de 88% à 63% (10/16) des processeurs, ce qui laisse par exemple suffisamment de processeurs libres pour un troisième utilisateur (Cf. figure 4.2).

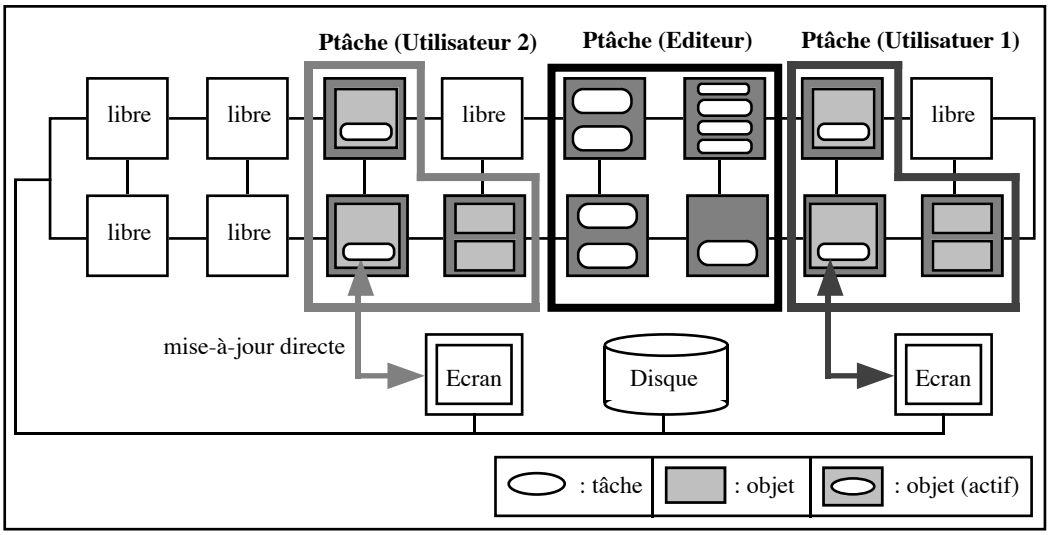


Figure 4.2. Deux éditeurs occupant 63% des processeurs

Ainsi, l’utilisation d’objets (actifs) a permis de résoudre de manière élégante le problème de la gestion des données séparées des flots d’exécution (processus) qui les manipulent.

4.1.2 De l'intérêt des objets

Un micro-noyau d'un système d'exploitation distribué est amené à être réparti sur tous les processeurs de la machine où le système est utilisé (ceci afin d'assurer la bonne gestion des communications, des processus, etc.).

D'autre part, les services de base (chargeur, gestionnaire de fichiers, serveurs de nom, etc.) sont aussi répartis et donc disposés sur un certain nombre de processeurs (par exemple, pour des raisons évidentes de performances, le SGF pourra n'occuper que les processeurs voisins des disques).

Ainsi le noyau (et aussi un grand nombre d'applications parallèles) se retrouve découpé en un certain nombre d'entités hétérogènes qui communiquent entre elles, et disposées non uniformément sur les processeurs de la machine. Ceci implique que :

- les structures de données doivent être définies avec attention, afin de permettre des accès parallèles et d'éviter les interblocages,

- Les algorithmes s'exécutant en parallèle, doivent donc verrouiller correctement les accès à toutes les structures de données partagées.

C'est pourquoi, encapsuler ces structures par des objets, avec uniquement appel de méthodes pour leur modification, permet de résoudre élégamment et relativement simplement ces différents problèmes.

En effet, afin de contrôler les accès concurrents sur une zone mémoire (disons une page), le système est par exemple obligé de maintenir, pour chaque page partagée, un verrou (simple, multiple, etc) qui permet de garantir la cohérence des données (par exemple, lorsqu'un thread modifie cette page, aucun autre thread ne doit lire son contenu). Malheureusement ceci implique une vision non uniforme de la mémoire (puisqu'on dispose de page avec verrou et de page sans verrous), et rend plus complexe l'écriture du noyau (nouvelles structures de données, gestion des verrous) et diminue ses performances (puisqu'il doit fournir plus de services). Enfin, le noyau ne peut fournir qu'un nombre réduit de mécanismes de synchronisation, ce qui peut pénaliser certaines applications aux besoins spécifiques.

Avec un objet, où les données ne sont accédées que par l'intermédiaire de méthodes, il est facile d'implanter un mécanisme de synchronisation des accès plus ou moins sophistiqué et personnalisé (Cf. chapitre **4.2.6 Synchronisation**). Le système n'a plus alors besoin de se préoccuper de la cohérence des pages, c'est l'objet qui les gère.

Pour des raisons d'efficacité, lorsque le noyau associe un verrou à une unité mémoire (page, segment, etc.), celui-ci contrôle l'accès d'une manière binaire : l'unité est soit verrouillée dans son intégralité, soit elle ne l'est pas. Or, le degré de parallélisme des accès est fortement dépendant de la granularité d'accès à la mémoire. Par exemple, si

l'unité mémoire est la page, un thread qui modifie une page peut bloquer un autre thread qui lit cette page, même si celui-ci n'accède pas la même portion de mémoire.

Avec un objet, qui peut être virtuellement de n'importe quelle taille, et qui connait (par définition) parfaitement sa représentation, on peut fournir un mécanisme qui permet de lire et de modifier simultanément son contenu.

Evidemment, il est possible que les performances globales de l'application s'en ressentent (plus le contrôle est complexe, et plus les temps d'accès augmentent), c'est pourquoi, les objets ne sont pas toujours la meilleure solution, cela dépend beaucoup de l'objectif initial (performance, fiabilité, etc.).

Un autre aspect intéressant de la programmation objet est la protection. En effet, à tout objet, on peut associer un certain nombre de droits (propriétaire, type d'accès, etc.) qui permettent de restreindre les accès aux données. Ces droits pouvant être contrôlés soit au niveau de l'objet lui même (par exemple pour vérifier si tel utilisateur à le droit d'utiliser l'objet) soit au niveau des méthodes (par exemple pour contrôler des accès en lecture/écriture).

Ainsi, dans beaucoup de noyaux (et c'est le cas dans PARX), les threads d'une même tâche sont sans protection, aucune les uns par rapport aux autres. Tous les threads d'une tâche ont accès à toutes les ressources de la tâche. Or certaines applications peuvent nécessiter une protection entre leurs différentes entités de calcul. Le fait de disposer d'objets qui représentent et gèrent les différentes ressources de la tâche permet de résoudre efficacement ce problème. Cette fonctionnalité introduisant un surcoût dans la gestion des threads, il n'est pas sûr qu'elle soit toujours la meilleur solution. Cependant, pendant une phase de mise au point par exemple, elle peut permettre de détecter plus rapidement un thread dont le comportement est erroné.

Enfin, et c'est un aspect non négligeable, les objets permettent de résoudre le problème de la gestion des variables globales d'un programme parallèle, particulèrement en évitant les conflits d'accès.

En conclusion, nous pourrons trouver bien des exemples (gestionnaire de fichiers dans lequel tout fichier est un objet actif, etc), où la notion d'objet rend de grands services. Aussi, il nous a paru utile de l'introduire dans le modèle d'exécution de PARX. Cependant nous avons aussi vu que la programmation objet ne doit pas devenir une contrainte en étant le seul modèle de programmation.

D'où la notion de *modèle hybride*. Celui-ci n'est ni un pur modèle à base de processus, ni un pur modèle à base d'objets. L'utilisateur est libre de choisir pour la conception de son application, grâce éventuellement à un langage de programmation spécifique, la représentation parallèle qu'il juge la plus adaptée à ses besoins : on ne programme pas à priori de la même manière une application scientifique qui travaille sur des matrices, ou un serveur parallèle de gestion de base de données dans un système réparti.

Cette approche hybride est utilisée dans d'autres systèmes comme par exemple le SDO Eden [ABLN85], où l'utilisateur peut à la fois utiliser les fonctionnalités d'Unix et les services d'Eden.

4.2 Le modèle à objets

Dans cette partie, nous décrivons les différents choix qui se sont imposés lors de la définition du modèle à objets de PARX.

En particulier, le modèle choisi est un modèle sans héritage. L'héritage est un mécanisme qui permet de développer de nouveaux objets à partir d'objets existants simplement en spécifiant comment les nouveaux objets diffèrent des originaux. C'est normalement une fonctionnalité offerte au niveau du langage. Cependant, nous pensons qu'un bon modèle à objets doit tenir compte de ce mécanisme, pour que la couche langage n'ait pas à tout réécrire (parce que par exemple, le modèle d'exécution est incompatible avec l'héritage, Cf. les points deux et trois du paragraphe suivant).

En dépit du fait que ce mécanisme d'héritage est très puissant et très attractif (il permet la spécialisation et la réutilisation du code), il présente un certain nombre de problèmes lorsqu'il est implanté sur des architectures distribuées [Am89] :

- Premièrement, comme les méthodes d'un objet peuvent être redéfinies dans un sous-objet et que nous ne pouvons plus contrôler statiquement les accès à un objet (car les canaux de communications entre processeurs distants ne véhiculent que des données non typées [BBK91]), il est nécessaire de disposer d'un mécanisme de sélection dynamique des méthodes, et d'un mécanisme de contrôle dynamique de type, ce qui entraîne une augmentation de la complexité dans la gestion des objets, et une possible baisse des performances du système.

- Deuxièmement, lorsqu'un objet migre (par exemple pour l'équilibrage de la charge dans un système parallèle), dans le pire des cas, tous ces "sur-objets" (les objets pères) doivent aussi être migrés, et les références (généralement des pointeurs) sur ces entités doivent être de nouveau recalculées. Conclusion, le coût de migration d'un objet peut s'avérer extrêmement important.

- Troisièmement, lorsque l'accès à cet objet se traduit par l'appel à une méthode appartenant à un objet père non présent dans la mémoire, quelle procédure doit-on suivre ? Faut-il aussi charger l'objet père en mémoire, ce qui risque de l'encombrer inutilement, ou bien faut-il envoyer un message vers un autre processeur distant qui lui contient ce sur-objet ? Dans ce cas, le temps d'exécution d'une méthode peut devenir anormalement long (d'autant plus que le sur-objet visé peut lui aussi envoyer un message vers un autre sur-objet pour exécuter la méthode), ce qui peut être très pénalisant pour l'application.

En raison de ces aspects, nous avons donc décidé de laisser provisoirement de côté le mécanisme d'héritage. Cependant, des recherches seront ultérieurement effectuées dans le but d'étudier la faisabilité de ce concept dans un contexte de parallélisme massif.

4.2.1 Granularité

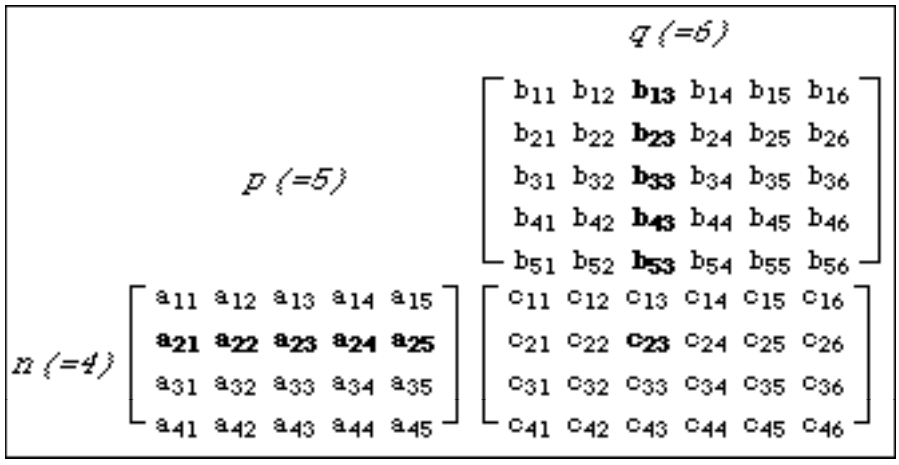
Comme nous voulons gérer des entités à la fois relativement grosses (comme des fichiers, des documents, des matrices, etc), et relativement petites (comme des listes chaînées, des éléments d'une matrices, etc.) il paraît normal de s'orienter vers un système qui supporte des *objets à gros grains*, et des *objets à grains intermédiaires*.

En revanche, en raison de l'approche modèle hybride choisi, qui permet donc une programmation à la fois standard et à objets (comme C++ par exemple [St87]), nous pensons que les objets à *grains fins*. comme les types de base du langage (entier, booléen, etc) et les "petits" objets en général, qui sont locaux à une exécution d'un processus, doivent être directement gérés par les compilateurs, ceci afin de réduire le nombre d'objets gérés par le système.

Cette technique a d'ailleurs été choisie dans de nombreux SDO tels que par exemple Guide [De90], SOS [SGHM89], et Cool [HMA90].

Mais le grain d'un objet n'est pas la seule notion intéressante dans un système parallèle. Considérons par exemple la situation suivante : supposons que l'on dispose d'une machine parallèle sans mémoire commune, chaque processeur ayant 1 Mo de mémoire dont 200 Ko occupés par le système (i.e. 800 Ko sont disponibles pour les applications), et supposons que nous ayons besoin de calculer le produit C de deux matrices $A=[a_{ij}]_{nxp}$ et $B=[b_{ij}]_{pxq}$, avec $(n,p,q)=(4,5,6)$.

Alors $C=[c_{ij}]_{nxq}$ est donné par : $c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}$, soit graphiquement :



En supposant que l'on travaille sur des éléments qui font 20 Ko (par exemple des images), on obtient : taille(A) = 400 Ko, taille(B) = 600 Ko, taille(C) = 480 Ko.

On pourrait donc avoir par exemple, les placements suivants :

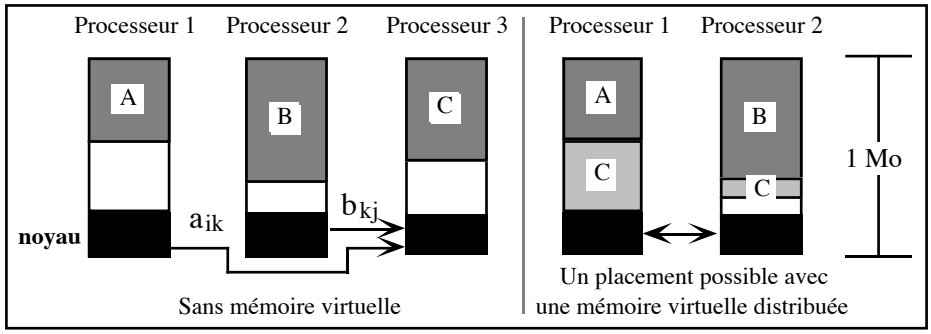


Figure 4.3. Placement d'objets (matrices) sur des processeurs

On se rend compte évidement que ces solutions présentent l'inconvénient de générer un grand nombres de messages (envoi de données de 20 Ko pour chaque élément de la matrice) entre les différents processeurs.

Heureusement cet algorithme est parallélisable. Il suffit de remarquer que `C = A x B`, peut aussi s'écrire : $\forall j \in$ `[1..q], C[j] = A x B[j]` où les `B[j]` (respectivement `C[j]`) sont des matrices colonne extraites de `B` (respectivement de `C`) [Ch92].

Alors si les objets matrices `B` et `C` peuvent être divisées en sous-objets que nous appelerons **fractions**, le placement devient celui de la figure 4.4 (les fractions de `B` sont les matrices colonne `B[j]`, et taille(`A`) = 400 Ko, taille(`B[j]`) = 100 Ko, taille(`C[j]`) = 80 Ko). Nous utiliserons la notation "`OBj`"`:i` pour désigner la ième fraction d'un objet "`OBj`".

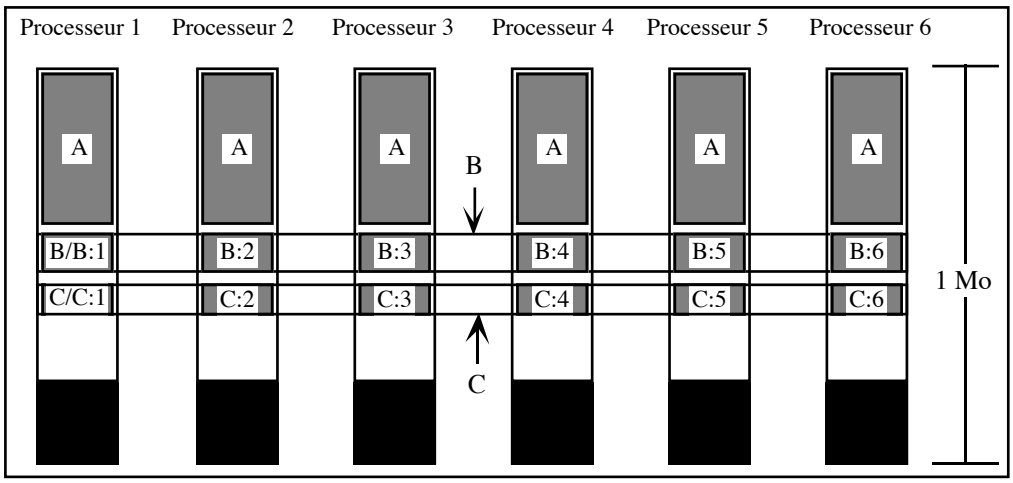


Figure 4.4. Placement de fractions d'objets (matrices) sur des processeurs

Les objets fractionnés permettent donc une gestion plus fine des applications en augmentant le degré le parallélisme des accès [CCL89], tout en permettant une diminution notable des transferts de données entre processeurs.

Définition d'un objet fractionné

Un *objet fractionné* est un objet qui a été découpé en plusieurs sous-objets de taille quelconque. Chaque partie est appelée une *fraction d'objet* ou *f-objet*.

La décision de découper un objet en plusieurs fractions et de placer les fractions sur les processeurs est *uniquement* prise au niveau utilisateur (par exemple, au moyen d'instructions particulières intégrées dans le langage de programmation,), car seul le développeur de l'application est vraiment capable de savoir s'il a besoin ou non d'exploiter cette propriété des objets. Ceci permet de plus de connaître statiquement les identificateurs de chaque fraction, et donc de simplifier leur gestion au niveau du système (puisque par exemple tout message sera directement envoyé au fraction visé, sans passer par des intermédiaires).

Propriétés

Les objets fractionnés d'un même objet sont tous *indépendants* les uns par rapport aux autres, ce qui signifie par exemple, qu'une *f*-objet ne peut communiquer avec une autre *f*-objet. Un objet fragmenté est toujours relié à un *objet racine* (B/B:1 et C/C:1 dans l'exemple précédent). En particulier, son identificateur est dérivé de l'identificateur de l'objet racine.

Lorsque la décision est prise de détruire la "racine", tous ses *f*-objets associées seront aussi détruites. En revanche, la racine peut supporter la destruction d'un ou plusieurs de ses *f*-objets sans que cela affecte son comportement ou celui des autres *f*-objets (puisqu'il n'y a pas de dépendances entre elles). Cette propriété est extrêmement intéressante dans le cas où par exemple, le système à besoin de faire de la place en mémoire, et sait qu'une fraction n'est plus référencée (par exemple parce que l'utilisateur l'aura lui-même désallouée). Il est aussi possible de spécifier que la destruction d'une *f*-objet implique la destruction de la totalité de l'objet (afin d'éviter par exemple d'éventuels problèmes de cohérence, si l'objet doit être sauvegardé sur disque).

Notes

On pourra constater que cette définition d'objet fractionné est très différente de la notion d'objet fragmenté utilisée dans SOS [SGHM89] où par exemple les différents fragments d'un objet peuvent communiquer entre eux. Nous pensons cependant que l'approche choisie dans SOS est plus complexe et ne s'adapte pas efficacement dans un environnement à parallélisme massif. Elle peut entraîner en effet des inconvénients que nous voulons éviter, comme un surcoût au niveau des performances (à cause des "talons" qui relaient les messages et qui donc augmentent le temps des communication, et de la migration des mandataires qui surchargent à la fois le réseau et le processeur sur

lequel ils sont migrés) et l'introduction de problèmes éventuels de cohérence (quand les objets fragmentés conservent en mémoire locale des copies de données distantes).

4.2.2 Visibilité d'un objet

Alors qu'un objet serveur de nom, ou gestionnaire de fichiers devra être normalement accessible par n'importe quelle autre entité système ou utilisateur (on dira alors que la *visibilité de l'objet* n'est pas limitée), les objets d'une application parallèle ne devront être à priori manipulés qu'au sein de la Ptâche correspondante (leur visibilité est restreinte à la Ptâche).

Pour réfléter cette différence, nous introduisons la notion d'*objet privé*, qui n'est visible uniquement que par les tâches, threads, etc d'une Ptâche i.e. sa visibilité est limitée à la Ptâche (on dit alors qu'il ne peut être *exporté*), et d'*objet public*, qui peut être visible à priori par n'importe quelle entité dans la machine (à condition bien sûr que l'objet se soit fait connaître du système).

La notion d'objets privés évite au système de les gérer directement (i.e. à garder leurs traces dans des tables internes) en transférant leur administration au niveau de la Ptâche (Cf. § **Gestion des entité**s dans le **Chapitre 5**). D'autre part, les temps d'accès aux objets privés seront généralement meilleurs puisque, par exemple, il faut véhiculer moins d'informations pour les localiser (Cf. § **Designation** dans le **Chapitre 5**). Enfin, un objet privé ne peut être déplacé qu'à l'intérieur de la Ptâche auquel il appartient.

Propriétés :

• Un objet est par défaut privé, mais cet état est modifiable lors de sa déclaration.
• Un objet public ne peut pas devenir privé, un objet privé ne peut pas devenir public.

4.2.3 Référence à un objet

Le fait de stocker ses données dans un objet n'apporte pas toujours que des avantages. Considérons par exemple, la situation suivante (Cf. figure 4.5) : plusieurs threads d'une même tâche accède aux données de cette tâche par l'intermédiaire d'un objet. Si l'objet est placé sur un autre processeur (par exemple pour des raisons d'insuffisance de mémoire), les performances globales de la tâche vont grandement en souffrir.

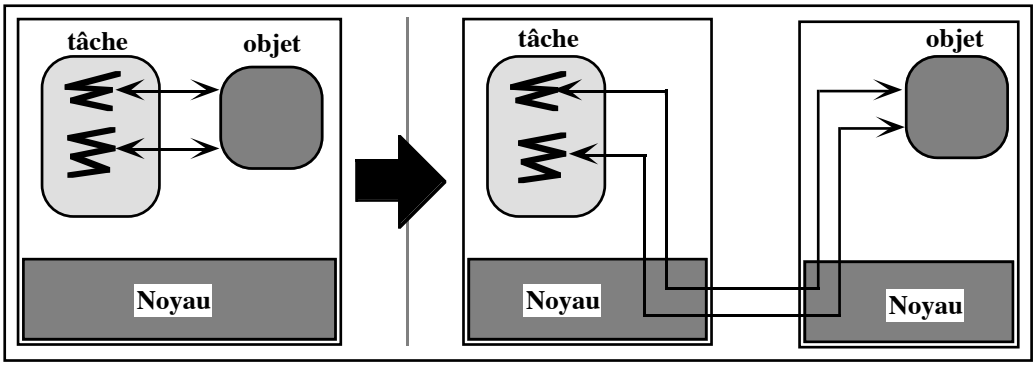


Figure 4.5. Surcoût de communication à la suite d'une migration d'un objet

On aimerait bien pouvoir indiquer au système que l'objet ne doit pas être déplacé (migré) seul ou bien qu'il doit être lié à une tâche de manière permanente, de telle sorte qu'il sera toujours placé sur le même processeur que la tâche.

C'est pourquoi nous avons introduit trois concepts qui permettent de répondre à ces problèmes :

- la notion de référence vers un *objet adhérent* ("*sticky*" en anglais), qui permet d'assigner de façon permanente un objet à un processeur, l'empêchant ainsi d'être déplacé.

- la notion de référence vers un *objet statique* qui lie l'objet à l'entité qui l'a créé ou aux entités avec lesquelles il communique. La ou les liaisons doivent être connues à la compilation, et une référence à cet objet ne peut être communiquée. Un objet statique ne peut donc jamais être déplacé de façon autonome, ni être invoqué à distance. Conclusion, nous pouvons utiliser un simple appel de procédure pour l'exécution des méthodes d'un objet, au lieu de passer par le mécanisme standard (appel à un serveur d'entités pour localiser l'objet et envoi d'un message), ce qui permet d'optimiser fortement les communications entre entités. Cette notion d'objet statique est similaire à la notion d'objet local dans Emerald [BHLC87].

  Ceci a donc pour conséquence que la migration d'un objet statique entraîne obligatoirement la migration de toutes les entités qui lui sont rattachées, à condition bien sûr qu'aucune de ces entités ne soient dans l'état adhérent. Si c'est le cas, la migration ne peut être effectuée.

- la notion de référence vers un *objet global* qui désigne tout objet qui n'est pas statique i.e. la liaison entre objets et entités peut se faire entre processeurs distants. Par conséquent, les communications (requêtes ou appels de méthodes) se font par envoi de messages, et il faut rajouter une *vérification dynamique de conformité* pour les objets, c'est à dire une vérification de la conformité de l'interface de l'objet chargé, avec les déclarations du contexte utilisateur [BBK91]. De cette définition, nous pouvons déduire qu'un objet public (donc accessible par n'importe quel autre objet) est forcément global.

Des définitions de la visiblité et de la référence à un objet, nous tirons les correspondances suivantes :

Vis.\Référence	***statique***	***global***	***adhérent***
privé	possible	possible	possible
public	impossible	toujours	possible

Propriétés :

• De par sa définition, un objet fragmenté ne peut être que global et adhérent,
• Tout autre objet est par défaut statique, mais cet état est modifiable lors de sa création.

Nous donnons sur la figure 4.6, un exemple qui montre l'influence de ces différents attributs sur la communication entre entités.

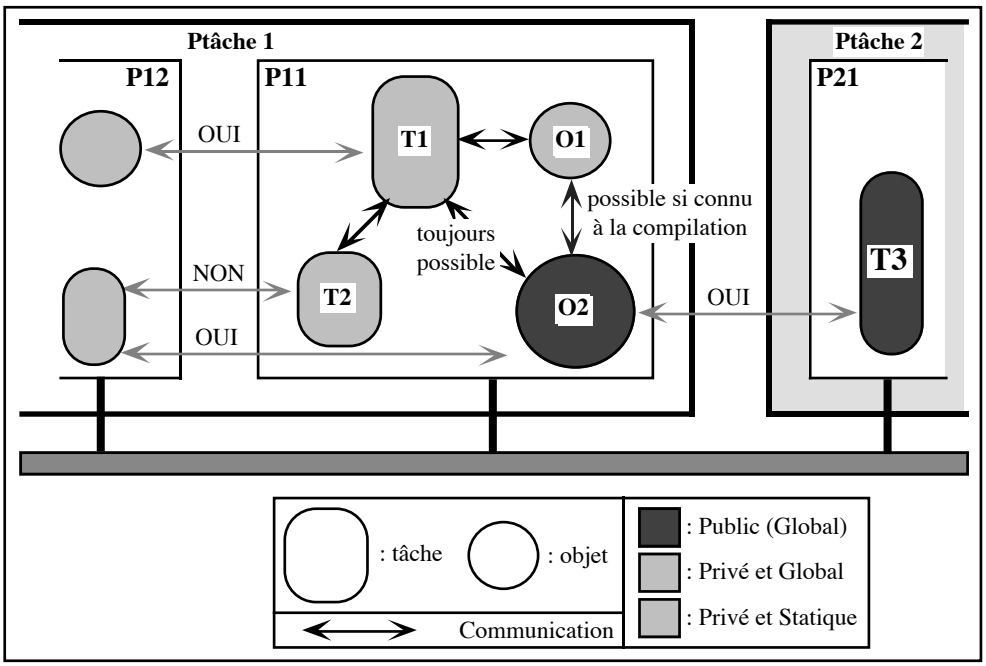


Figure 4.6. Visibité et référence aux entités.

Ainsi une tâche privée et statique (par exemple T2) ne sera non seulement visible qu'à l'intérieur de la Ptâche auquelle elle appartient, mais en plus, seules les entités localisées sur le même processeur (T1, 01, O2) pourront éventuellement communiquer avec elle. En revanche, l'objet O2 de la Ptâche P1 qui est public est visible par la tâche T3 de la Ptâche 2.

Note : on pourra remarquer que ces notions (visibilité et référence) ne sont pas l'apanage unique des objets. Ainsi, à chaque entité du système peuvent en effet être assignés les attributs suivants (un signe "-" dans une case indique que l'attribut est impossible) :

Entité	***Visiblité***		***Référence***		
	Public	**Privé**	**statique**	**global**	**adhérent**
Ptâche	non applicable		—	toujours	possible
tâche	possible	possible	possible	possible	possible
thread	—	toujours	toujours	—	possible
port	possible	possible	possible	possible	possible
canal	—	toujours	possible	possible	possible

4.2.4 Comportement d'un objet

Le modèle objet de ParObj est par défaut un *modèle à objet passif*. En particulier, les objets statiques étant par définition des objets à faible coût d'accès, ceux-ci ne pourront qu'être passifs.

En revanche, dans le cas d'objets publics (en particulier au niveau système, comme par exemple un serveur de nom), la notion d'*objet actif* peut être nécessaire et utile [ABLN85], [Ag86], [AH87], [Ni87], [PK90], [FLM91], [Sc92] (par exemple ce même serveur de nom peut décider temporairement de sauvegarder ses informations sur disque, ou bien interroger d'autres serveurs de nom, etc.).

Les objets actifs sont des objets qui peuvent s'exécuter en parallèle avec d'autres entités indépendantes (tâches par exemple). Par exemple, un objet serveur ne doit pas rester dans un état bloqué s'il ne peut pas satisfaire immédiatement une requête. Si le thread associé à la requête est bloqué, la main est rendue à l'objet afin qu'il puisse éventuellement servir une autre requête. Lorsque le thread reprend son exécution, l'objet termine la requête associée.

Un certain nombre de *script d'arrière plan* peuvent aussi être attachés à un objet actif. Ces scripts, qui sont implantés par des threads dans PARX, peuvent être exécutés soit lorsque l'objet est au repos, soit après un intervalle spécifique de temps, soit à la suite d'événement particulier (une entrée/sortie par exemple). Ces scripts d'arrière plan peuvent être utilisés par exemple pour réaliser des totaux de contrôle de données sensibles, pour sauvegarder des objets en mémoire secondaire, etc.

Enfin, pour ne pas interrompre entièrement la machine lorsque le système est mis à jour, les objets (système) actifs doivent être capable d'accepter dynamiquement une modification de leur comportement (ceci évidemment après un contrôle des droits d'accès, afin d'éviter qu'un utilisateur astucieux n'en profite pour percer les protections du système).

Par exemple, si une erreur a été découverte dans une des méthodes de l'objet actif serveur de fichier, lors du remplacement dynamiquement de la méthode fautive, le système ne bloque temporairement que les entités qui accèdent au serveur de fichier, tout en laissant les autres entités (Ptâches, etc) fonctionner correctement. Dans un environne-

ment universitaire ou industriel, ou la disponibilité de la machine est un critère d'extrême importance, cette fonctionnalité devient très précieuse. On pourra noter que ce service s'intègre parfaitement au concept de *systèmes ouverts* (systèmes qui évoluent dynamiquement) prônés par [HAJ82].

Les objets actifs peuvent être une solution élégante au problème de la gestion des fichiers sur une machine parallèle. Ainsi chaque fois qu'un fichier est accédé (créé ou ouvert), le système crée un objet actif. Cette objet pourra être manipulé grâce à un certain nombre de méthodes comme `read`, `write`, `seek`, `stat`, etc, et disposera en plus d'un script d'arrière plan pour la sauvegarde périodique de ces données. Evidemment l'utilisateur pourra dynamiquement modifier le comportement du fichier en lui spécifiant par exemple l'intervalle de temps entre chaque sauvegarde, ou bien le nombre de blocs à charger en mémoire lors de chaque lecture sur disque…

4.2.5 Protection

Tout comme les processus de PARX sont protégés par des capacités, il en est de même pour les objets globaux. Notons qu'il n'est pas nécessaire de protéger un objet statique, puisqu'il ne peut être accédé que par les entités auxquelles il est lié, et que dans ce cas, le contrôle d'accès a pu être vérifié à la compilation.

4.2.6 Synchronisation

Par défaut, nous utilisons un *mécanisme de synchronisation pessimiste* : une requête qui invoque un objet est temporairement suspendue si elle interfère avec une autre requête qui est en cours de traitement par l'objet [CC91].

Un exemple typique est l'appel par deux processus différents (respectivement `P1` et `P2` dans cet ordre) de la même méthode `m1(nouvelle_valeur)` (d'un objet `Obj1`) qui modifie une variable interne (`R1`) :

P1	**P2**
`Obj1.m1(10)`	…
…	`Obj1.m1(20)`

Si aucun mécanisme de synchronisation n'est implanté, l'exécution (dans cet ordre) des deux opérations précédentes est *non-déterministe*. En effet, si `P1` après avoir appelé `O1.m1(10)` est endormi (par exemple à la suite de l'écoulement de son quantum de temps), `P2` peut alors avoir largement le temps d'exécuter `O1.m1(20)` avant que `P1` ne devienne actif de nouveau. Dans ce cas, la valeur finale de `R1` sera 10 (et non 20) ! En sérialisant les accès à `O1.m1()`, on est sûr que la valeur finale de `R1` sera bien 20.

La *sérialisation des accès* peut être réalisée par un simple verrou de contrôle en lecture/écriture : si le verrou est libre, la méthode peut être exécutée, dans le cas contraire,

l'appelant est mis dans une file d'attente. C'est par exemple ce qui a été réalisé dans le langage μC++ [BDSY92], où les méthodes d'un objet (toutes par défaut) sont sérialisées par une abstraction appelé uMutex.

Si aucune condition de synchronisation n'est attachée à une méthode, on dit alors que son exécution est *sans contrainte* (**Note** : C'est le comportement par défaut). Par exemple, un objet en lecture seule ne nécessite généralement pas de mécanisme de synchronisation.

Dans toute la suite de ce chapitre, nous utilisons un language à objets imaginaire, néanmoins fortement inspiré par des langages comme Guide, Sather et C++ (pour les constructeurs, et destructeurs) pour décrire les différents exemples.

Exemple 1 :

```
LOCK lockWrite ;

METHOD write(int fd, char *buf, long len) IS
      <écriture des données dans le fichier fd>
END ;

SYNCHRO write IS
   // EVAL évalue d'une manière atomique une expression
   // Si la méthode a été appelée, lockWrite est dans l'état
   // bloqué et l'évaluation de "EVAL lockWrite" retourne FAUX:
   // dans ce cas l'appelant est endormis. Si l'évaluation
   // retourne VRAI, la requête est servie (exécution de la
   // méthode correspondante).
   //
   // Evidemment, le verrou n'est relâché qu'après terminaison
   // de la méthode associée.
   EVAL lockWrite ;
END ;
```

Il doit être aussi possible de spécifier un ensemble de verrous pour synchroniser une même méthode. En effet, l'accès à une ressource peut être tributaire de la présence ou non d'autres ressources : supposons qu'un objet actif serve de relais entre deux ports série A et B. Ce processus n'est autorisé à faire un transfert que lorsque les deux ports sont disponibles. Dans ce cas, un verrou sera attaché à chaque port, son état étant bloqué si le port n'est pas disponible, non bloqué dans le cas contraire. Alors la procédure transfert pourra par exemple s'écrire de la manière suivante.

Exemple 2 :

```
// La condition "lockA AND lockB" est évaluée : si elle est
// vraie, la clause de synchronisation est donc vérifiée ; dans
// ce cas la méthode correspondante est exécutée. Dans le cas
// contraire, l'appelant est suspendu et la requête associée
// est stockée dans la file d'attente globale de l'objet.
```

```
METHOD transfert IS
      <transfert des données>
END ;

SYNCHRO transfert IS
   EVAL (lockA AND lockB);// LockA, LockB sont de type LOCK
END ;
```

Malheureusement, les verrous augmentent les temps d'accès aux objets en bloquant plus ou moins longuement les processus qui y accèdent. Par exemple, si une méthode contrôle l'accès à un tableau, chaque fois qu'un élément du tableau est modifié, toutes les requêtes à cette méthode sont bloquées, même si elles veulent accéder à des éléments différents du tableau. Dans ce cas, si le temps d'exécution est le critère prédominant de l'application, une solution, pour améliorer le temps de réponse d'un objet, consiste à utiliser des *f*-objets (objets fractionnés), permettant ainsi un plus grand degré de parallélisme (par exemple en découpant le tableau en un ensemble de sous tableaux).

Il existe aussi des situations dans lesquelles l'utilisateur a besoin d'implanter un mécanisme spécifique de synchronisation. Ainsi, lorsqu'un objet n'est pas prêt à rendre un service, il peut traiter la situation soit en retournant une erreur (et dans ce cas c'est à l'appelant d'essayer de nouveau un peu plus tard), soit en *différant* l'appelant jusqu'à ce que sa requête puisse être traitée.

Il est donc nécessaire de disposer d'un mécanisme plus sophistiqué que les simples verrous. D'autre part, pour écrire des schémas complexes de synchronisation, il apparaît plus facile (et plus logique) d'écrire un script (que l'on attache à la méthode correspondante) contenant des opérateurs de synchronisation et des évaluations d'expressions booléennes, que de mélanger le code et la synchronisation, rendant la compréhension de l'ensemble nettement plus obscure. Enfin, le fait de disposer d'un script séparé du corps de la méthode permet d'évaluer très facilement de nombreux protocoles de synchronisation, sans avoir à changer une seule ligne de code de la méthode [RV77].

Ainsi, l'un de ces mécanismes consiste à associer à toute méthode un ensemble de *compteurs de synchronisation* [RV77], [De90], [DDRR91]. Ces compteurs sont des variables internes qui spécifient pour chaque méthode : le nombre total d'invocations depuis l'initialisation de l'objet (`<méthode>.total`), le nombre total d'exécutions terminées depuis l'initialisation (`<méthode>.completed`), le nombre total d'invocations autorisées depuis l'initialisation (`<méthode>.authorized`). Ces trois compteurs sont initialisés à zéro à leur création, et ne peuvent qu'augmenter au cours du temps.

De ces trois compteurs, on peut facilement déduire le nombre de requêtes en cours d'exécution :

```
<méthode>.current = <méthode>.authorized - <méthode>.completed
```

et le nombre courant de requête en attente :

```
<méthode>.waiting = <méthode>.total - <méthode>.authorized
```

L'exemple 1 peut aisément s'écrire de la manière suivante (c'est d'ailleurs la solution la plus efficace):

Exemple 3 :

```
METHOD write(int fd, char *buf, long len) IS
      <écriture des données dans le fichier fd>
END ;

SYNCHRO write IS
   EVAL ( write.current == 0 ) ;
END ;
```

Bien que cet exemple semble démontrer l'inutilité des verrous, nous pensons que ce constructeur à son utilité, d'autant plus que, à l'opposé des compteurs de synchronisation qui ne sont pas modifiables, il est possible de déverrouiller explicitement un verrou (`<verrou>.unlock`). Cette opération est par exemple très utile dans l'exemple suivant (si le nombre d'invocations de `m1` est très supérieur à `m2`, et que le temps d'exécution de `m2` est grand par rapport à `m1`, la possibilité de relâcher le verrou 1 avant l'exécution de `m2`, permet d'augmenter fortement le taux de réponse de `m1`) :

Exemple 4 :

```
METHOD m1 IS    // corps de la méthode non détaillé
END ;

METHOD m2 IS    // corps de la méthode non détaillé
END ;

SYNCHRO m1 IS
   EVAL Lock1 ;
END ;

SYNCHRO m2 IS
   EVAL ( Lock1 ) THEN
      // le verrou 1 a été acquis
      EVAL ( Lock2 ) THEN
         // le verrou 2 a été acquis
         Lock1.unlock ;
      END ;
   END ;
END ;
```

Enfin nous avons aussi vu qu'il était souhaitable de pouvoir explicitement différer un appel. [Ni87] montre que ce mécanisme peut être réalisé grâce à des *files d'attente à retard* ("*delay queue*"). Une telle file est soit dans l'état *ouvert*, et dans ce cas les requêtes sont acceptées, soit *fermé* (les requêtes sont mises dans la file d'attente), et l'uti-

lisateur a la possibilité de modifier ces deux états (en faisant respectivement un `queue.close` ou un `queue.open`). La forme la plus simple de synchronisation avec file d'attente est la suivante :

Exemple 5 :

```
// On indique au compilateur que les processus bloqués doi-
// vent être mis dans la file aQueue et non dans celle de l'objet
METHOD aMethod USING aQueue IS
END ;

La méthode <aMethod> sera exécutée si la file <aQueue> est ouverte,
sinon l'appelant sera suspendu.
```

Il est aussi parfois utile de pouvoir remettre à zéro une file d'attente (par exemple lorsqu'on veut débloquer un ensemble de processus, ou lorsque l'objet qui détient la file d'attente est détruit, etc), ou bien d'interroger son état. Le langage pSather [FLM91] par exemple, définit des "sortes" de moniteurs dans lesquels ce type de manipulations est possible. Nous pensons que ces opérations permettent une plus grande souplesse dans l'écriture des conditions de synchronisation. C'est pourquoi, les opérateurs suivants ont été introduits : `queue.clear` permet de vider la file des requêtes, et de débloquer ainsi les processus appelants (*mais avortant l'exécution de la méthode*), tandis que `queue.abort(<exception_type>)` a le même effet que `queue.clear` mais avec l'envoi d'une exception en plus. L'interrogation de l'état de la file se fait par l'appel à `queue.is_closed`, `queue.is_open` (qui retournent un booléen), et `queue.nb_threads` (qui retourne le nombre de threads bloqués dans la file).

L'introduction des scripts de synchronisation rend très certainement plus complexe la gestion des méthodes d'un objet. Cependant, nous pensons que leur introduction était justifiée. Le lecteur pourra se reporter à l'**Annexe C** pour une étude de l'algorithme d'exécution d'une méthode.

A titre d'exemple, la réalisation d'une *N-barrière* (outil qui permet à N processus de se synchroniser) peut être écrite de la manière suivante :

Exemple 6 :

```
OBJECT CBarriere IS
  DQUEUE  q ;       // déclaration d'une file d'attente à retard
  int     N ;

  METHOD +CBarriere (int N) IS       // constructeur
     SELF.q = new DQUEUE(closed);    // initialement fermée
     SELF.N = N                  ;
  END ;

  METHOD -CBarriere IS          // destructeur de l'objet
```

```
    // Note : si q contient encore des processus, alors un
    // q.abort(<obj_deleted>) sera émis.
    dispose(q) ;
  END ;

  METHOD stop IS
    // vide ! (De toute façon n'aurait jamais été exécutée
    // (à cause du q.clear dans le script de synchronisation)
  END ;

  SYNCHRO stop USING q IS
    EVAL ( q.nb_threads == 0 ) THEN
      q.close ;
    ELSE
      EVAL ( q.nb_threads == N ) THEN
        // On débloque les N processus en attente, mais ils
        // n'exécuteront pas la méthode.
        q.clear ; // q.nb_threads est donc remis à zéro
      END ;
    END ;
  END ;
END CBarriere ;
```

la barrière pourra alors être utilisée de la manière suivante :

```
#define N       100          // ou toute autre valeur
#define M(i)    ((i) % N)    // i modulo N

main IS

  GLOBAL CBarriere CB = new CBarriere(N) ;
  GLOBAL CIntArray CI = new CIntArray(N) ;

  // lancement de N processus en parallèle qui initialisent
  // aléatoirement le tableau CI, puis se synchronisent avant
  // de calculer la moyenne de chaque élément.
  PAR [N] WITH i
    CI[i].put(random(i))  ;
    CB.stop               ;

    printf("%d\n",(CI[M(i-1)].get + CI[M(i+1)].get)/2);
  END ;

  dispose(CB) ;
  dispose(CI) ;
END ;
```

4.2.7 Persistance

Les objets passifs sont toujours volatils. Les objets actifs sont aussi volatils par défaut. Cependant il est possible de les rendre persistants en créant un thread d'arrière plan qui se charge de sauvegarder l'objet à intervalle régulier. Cela peut être le cas pour par exemple certains objets actifs du système, comme le gestionnaire de fichiers, les programmes d'impression désynchronisés ("spooler"), etc.

Chapitre 5 : Gestion des entités dans ParObj

5.1 Support de base pour la gestion des entités

Le but de ce chapitre est la définition d'un modèle d'exécution pour le système distribué à objets ParObj. En fait, c'est la partie système distribué qui est abordée ici, c'est à dire toutes les techniques qui permettent la gestion des entités à l'exécution.

Nous avons essayé de définir une architecture de système complètement indépendante de la machine cible. D'autre part, nous avons volontairement laissé de côté certains aspects du système distribué, comme la migration des entités, le placement des entités sur les différents processeurs, et l'équilibrage de charge qui font l'objet d'intenses recherches au sein de l'équipe [El92], [TM90], [TM92]. Cependant, un certain nombre de mécanismes ont été envisagés, qui devraient simplifier cette tâche d'intégration.

L'avènement des machines parallèles sans mémoire commune pose un certain nombre de problèmes lorsqu'on conçoit un système d'exploitation (distribué). En effet, en plus de la gestion des ressources locales à chaque processeur (mémoire, etc.), il faut aussi gérer les processeurs en tant que ressources du système. En particulier, lorsqu'on lance une application, doit-on privilégier le temps d'exécution des applications (en lui donnant le maximum de processeurs possible) empêchant peut être l'exécution d'autres applications, ou doit-on au contraire privilégier le taux d'efficacité du système en maximisant le nombre d'applications exécutées dans un intervalle de temps donné ?

D'autre part, comment doit-on gérer la mémoire distribuée ? Est-ce que les techniques classiques de "swap" par exemple sont encore utilisables ? Enfin, que devient la gestion des fichiers dans une telle architecture ? Nous avons donné une solution possible dans le **Chapitre 4** avec des objets actifs, mais par exemple, comment doit-on les stocker sur disque ?

Nous n'allons pas tenter, dans ce paragraphe, de répondre à tous ces problèmes. Aussi, nous nous bornerons à faire des hypothèses sur l'existence de telle ou telle fonctionnalité et dans le meilleur des cas, nous nous permettrons de proposer des fonctionnalités qui seraient utiles au système.

5.1.1 Ordonnancement

Il faut d'abord distinguer l'ordonnancement entre Ptâches, de l'ordonnancement à l'intérieur d'une Ptâche. L'ordonnancement entre Ptâche va caractériser le temps de réponse de tout le système, alors que l'ordonnancement à l'intérieur d'une Ptâche va déterminer ses performances. Malheureusement ces deux notions ne sont pas complètement indépendantes. Par exemple, allouer un petit nombre de processeurs à chaque application permettra un meilleur débit du système mais affectera sûrement les performances des applications, et vice versa.

En dépit du nombre assez restreint d'études dans ce domaine, les recherches semblent converger vers la nécessité de disposer d'informations sur l'application parallèle (Ptâche) à lancer pour définir un bon algorithme d'ordonnancement. Certains ont besoin de connaître le *grain de parallélisme* de l'application [MEB88], d'autres utilisent la *signature* d'un programme parallèle (qui est la vitesse à laquelle un programme s'exécute en l'absence de tout autre programme, et qui dépend du nombre de processeurs alloués, du type d'architecture, et de la façon dont a été réalisé le programme [DO88], [GST90], [GSTN90], [PD89]). D'autres enfin considèrent le *profil* (qui décrit en fonction du temps le nombre de processeurs utilisés par l'application) et la *forme* (qui donne le pourcentage de temps utilisé pour un nombre varié de processeurs) d'une application parallèle [Se89].

De toutes ces études, il ressort que le nombre de processeurs alloués est un critère pertinent et suffisant pour les algorithmes d'ordonnancement. Le problème consiste alors, dans un environnement multi-utilisateurs, à tenter d'allouer un nombre (optimal) de processeurs à chaque Ptâche pour qu' elle s'exécute le plus rapidement possible, tout en essayant d'avoir le plus grand nombre possible de Ptâches qui s'exécutent en parallèle dans le système.

Certaines études montrent que ce problème peut être résolu en allouant statiquement (i.e. au lancement) les processeurs à la Ptâche ([GST90], [Se89]) alors que d'autres prouvent que seule l'allocation dynamique où le nombre de processeurs dédiés à une Ptâche varie en cours d'exécution ([MEB88], [PD89]) peut résoudre le problème.

Mais le vrai problème réside dans la détermination du nombre optimal **`p-opt`** de processeurs. En fait, en fonction des objectifs (privilégier le temps de réponse du système ou le temps d'exécution des Ptâches) la valeur de `p-opt` n'est pas la même. Nous devons donc calculer un `p-opt` qui optimise le taux de réponse et un `p-opt` qui optimise le temps d'exécution**.**

Par la suite, nous appellerons **`maxp`** le nombre idéal de processeurs qui optimise le temps d'exécution de l'application et **`pws`** le nombre idéal de processeurs qui permet un compromis entre exécuter la plus rapidement chaque Ptâche, et garantir un temps de réponse optimal de la part du système. Cette dernière valeur est appelé *partie active de processeurs* ("processor working set" en anglais, d'où le nom) et on peut trouver dans [GST90] une manière de la calculer.

A ces deux valeurs, nous en ajoutons une troisième qui est le nombre minimal de processeurs au dessous duquel l'application parallèle ne doit pas être lancée (et que nous appellerons **`minp`**). Cette information pouvant être obtenue soit explicitement grâce par exemple à un mot clé du langage, soit implicitement grâce à des outils spécifiques (comme les analyseurs de programmes [Eu90], [An90]).

Nous faisons l'hypothèse qu'il est toujours possible, même approximativement, de calculer pour une application parallèle, les trois valeurs `minp`, `maxp` et `pws`. Ces trois valeurs, qui font parties des attributs d'une application parallèle, seront utilisés par le système pour déterminer l'ordonnancement des Ptâches, pour optimiser leur gestion mémoire, et pourront entrer en compte dans les algorithmes d'équilibrage de charge et de migration.

Il faut cependant bien reconnaître que si la détermination de `minp` est relativement aisée, celle de `maxp` et `pws` demeure relativement ardue. En effet, tout le calcul repose sur l'évaluation de la signature d'une application parallèle, et cette valeur ne peut généralement être obtenue que par l'intermédiaire d'heuristiques, de modèles probabilistes [Me88], ou par l'analyse à posteriori de l'exécution de l'application parallèle [HE91].

Note : il est possible d'envisager des applications déjà compilées (issues d'autres environnements par exemple), qui ne possèdent pas ces trois paramètres. Dans ce cas, le système doit être capable de leur calculer un triplet. Pour cela, à chaque fois qu'une Ptâche est lancée, le système recalcule un `maxp` et un `pws` moyen grâce à la formule suivante (`n` est le nombre total de Ptâches lancées depuis le démarrage de la machine) :

$$\overline{\mathbf{maxp}} = \frac{\sum_{i=1}^{n} \mathtt{maxp}_i}{n}, \quad \overline{\mathbf{pws}} = \frac{\sum_{i=1}^{n} \mathtt{pws}_i}{n}.$$

Il suffit alors de prendre pour le `minp`, `maxp`, `pws` les valeurs :

$$\mathtt{minp} = 1,\ \mathtt{maxp} = \overline{\mathtt{maxp}},\ \mathtt{pws} = \overline{\mathtt{pws}}.$$

5.1.2 Gestion mémoire

Nous ne faisons aucune hypothèse particulière sur la manière dont est gérée la mémoire. Il est certain que par exemple disposer d'une mémoire virtuelle distribuée peut présenter certains avantages (uniformité des accès au niveau utilisateur, …, Cf. [Ca91] pour plus d'information), mais nous pensons que le système doit être conçus sans supposer l'existence de telle ou telle fonctionnalité.

Cependant, quelque soit le type de gestion mémoire implantée, une difficulté fondamentale demeure : le problème du "*swap*" c'est-à-dire le vidage sur disque de certaines données en mémoire afin de gagner de la place pour en installer de nouvelles. Or, si dans les systèmes traditionnels à architecture monoprocesseur, cette technique n'est pas trop pénalisante, elle peu devenir extrêmement problématique sur des machines parallèles, lorsque la mémoire qui doit être purgée doit éventuellement traverser un certain nombre de processeurs avant de se retrouver sur le disque.

Nous ne savons pas actuellement comment ce problème sera résolu, nous pensons cependant qu'un moyen d'y remédier consisterait à créer un ou plusieurs serveurs mémoire (*Ptâche-mémoire*) qui feront office de cache pour le disque. Nous pourrions alors établir une sorte de hiérarchie de la mémoire (mémoire primaire, secondaire, tertiaire, etc.) qui pourrait être par exemple exploitée par les algorithmes d'ordonnancement entre tâches d'une Ptâche. Ces zones de mémoire devenant d'autant plus intéressantes que les Ptâches qui s'exécutent sur la machine parallèle sont allouées avec de moins en moins de processeurs (par exemple parce que la machine est devenue trop chargée).

Chaque Ptâche-mémoire devra sûrement être complètement autonome et le nombre de processeurs qui la constituent pourra évoluer au cours du temps. Nous pouvons même imaginer que lorsqu'une application est lancée, le système alloue un certain nombre de processeurs d'une Ptâche-mémoire pour la gestion du swap de cette tâche. Ce nombre pouvant être par exemple une fraction du nombre de processeurs attribués à la Ptâche.

5.1.3 Gestionnaire de fichiers

L'étude d'une système gestionnaire de fichiers sur machine parallèle fait actuellement l'objet d'une thèse au sein de l'équipe [Ca91]. Nous admettrons donc qu'un tel système existe et nous ne nous préoccuperons pas de savoir comment il fonctionne. Nous supposerons donc que les opérations classiques telles que `open()`, `create()`, `read()`, `write()`, etc. sont disponibles.

5.2 Architecture du système ParObj

5.2.1 Organisation générale

Nous avons vu, dans le **Chapitre 3** (Architecture du noyau PARX) que sur chaque processeur de la machine parallèle s'exécute le micro-noyau qui offre différents niveaux de machines virtuelles. Au dessus, des sous-systèmes fournissent un environnement de développement et d'exécution des applications, chaque système mettant en œuvre un ensemble de politiques qui lui est propre, à partir des mécanismes de base fournis par ce µ-noyau. ParObj fait partie de ces sous-systèmes.

Chaque sous-système est constitué d'un certain nombre de clusters sur lesquels s'exécutent des Ptâches dont le rôle est la gestion et l'administration des Ptâches utilisateurs de la machine.

Dans PARX, un cluster fournit un domaine de communication dans lequel chaque processeur dispose d'une table de routage qui lui permet d'envoyer un message vers n'importe quel autre processeur du cluster. Ceci étant, en dehors du cluster, il n'est pas toujours possible de trouver un chemin pour atteindre n'importe quel processeur de la machine (c'est le cas par exemple, pour les machines de type Supernode). Dans ce cas, il est fort possible qu'à la suite d'une reconfiguration de la machine, deux Ptâches quelconques ne puissent pas ou plus communiquer entre elles, Cf. figure 5.1, ce qui peut s'avérer relativement problématique !

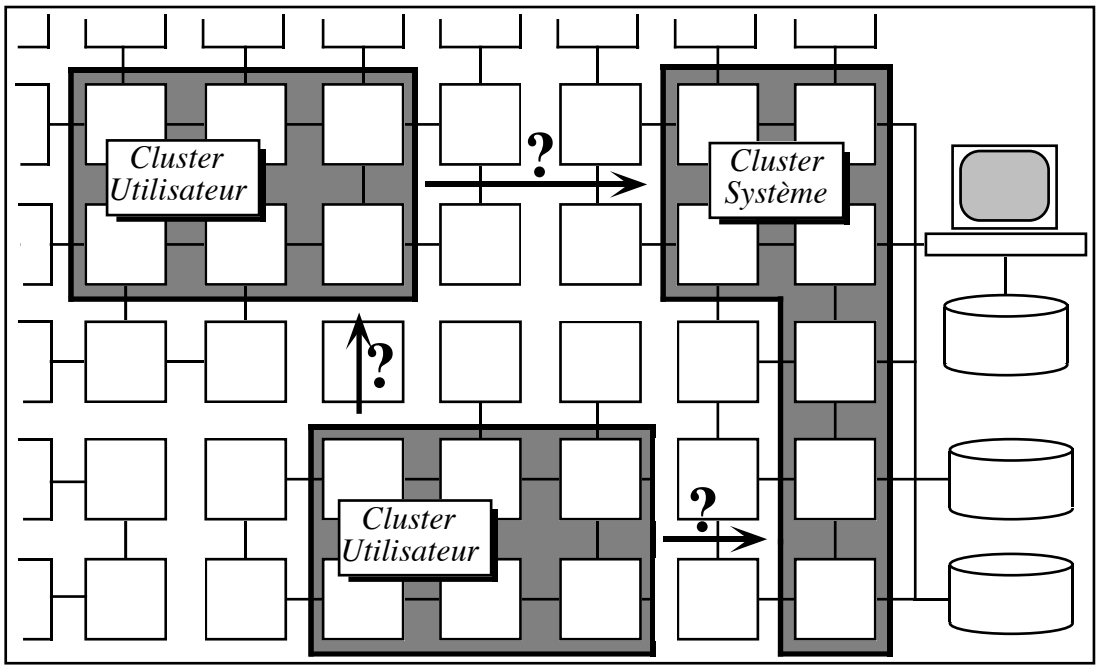


Figure 5.1. Problème de communication entre cluster

Évidemment sur des machines à topologies fixes (i.e. non reconfigurables) ces problèmes ne se posent pas, puisque n'importe quel processeur de la machine peut être atteint. Mais comme nous devons tenir compte de toutes les architectures, nous devons aussi traiter le cas des machines dynamiquement reconfigurables.

Le seul moyen alors pour faire communiquer deux clusters, indépendamment de l'architecture, et de disposer de un ou plusieurs processeurs sur chaque clusters et qui établissent un *pont de communication* entre les clusters (ces ponts, que nous appelons *P-connecteurs*, peuvent éventuellement changer de place au cours du temps, au fur et à mesure que des nouveaux clusters se créent ou disparaissent).

Ainsi tout message qui sort du cluster devra passer par ce ou ces processeurs, qui sont les seuls à avoir dans leurs tables de routages les informations nécessaires pour envoyer le message en dehors du cluster, Cf. figure 5.2.

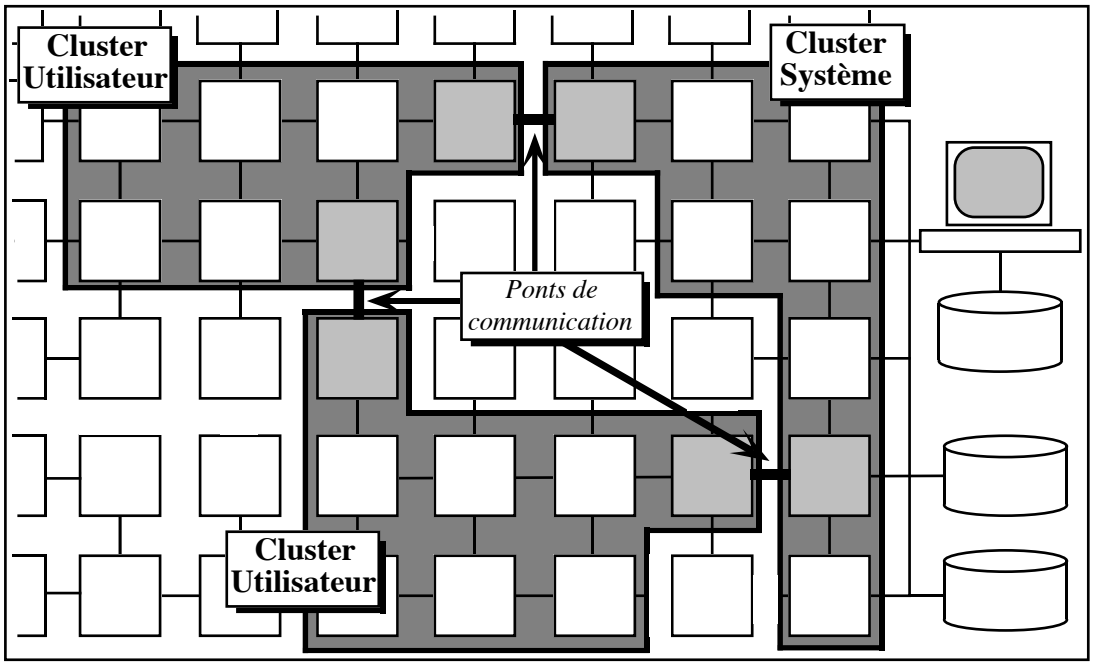


Figure 5.2. Ponts de communication entre cluster

Nous avons donc résolu nos problèmes de communication. Cependant la solution proposée n'est pas satisfaisante pour plusieurs raisons :

- Les Ptâches utilisateur (qui sont sur les clusters utilisateurs) ont directement accès, via le pont de communication, au cluster système, ce qui peut entraîner des problèmes de protection du système.

- Chaque Ptâche utilisateur a besoin d'être contrôlée (pour garantir sa bonne exécution). Cela signifie que le système doit assigner un certain nombre de processeurs de son cluster à la réalisation de ce travail. Conclusion, plus le nombre de Ptâches va croitre, et plus le système risque de réserver de processeurs pour leur gestion, entraînant un temps de réponse de plus en plus lent de la part du système (soit

parce qu'il aura de moins en moins de processeurs pour ses propres activités, soit, dans le cas de machines reconfigurables, parce qu'il passera son temps à reconfigurer son cluster en y ajoutant régulièrement des processeurs, paralysant ainsi toutes ou une partie des communications entre lui et le reste de la machine).

Nous pensons donc qu'il est nécessaire d'introduire un nouveau cluster système dont le rôle sera la gestion des différentes Ptâches utilisateurs et la communication entre toutes les Ptâches (utilisatrices ou système) de la machine.

Cette idée d'une interface entre le système et les utilisateurs n'est pas nouvelle. Ainsi, par exemple dans PEACE [Sc90], chaque cluster de la machine SUPRENUM est constitué de plusieurs nœuds (un ou plusieurs processeurs) dont un certain nombre est réservé pour l'interface entre des disques, pour établir la communication entre les clusters (appelés *nœuds de communication*), et pour réaliser des diagnostics.

Ce cluster ayant pour rôle la gestion des Ptâches utilisateurs, l'interface avec le système et la communication entre Ptâches, nous l'appellerons *cluster de contrôle*. Ce cluster n'étant constitué que d'une seule Ptâche, nous la désignerons sous le nom de *Ptâche de contrôle*.

S'il existe un ou plusieurs *cluster mémoire* dans la machine, ils devront avoir des ponts de communication non seulement avec le cluster système, mais aussi avec le cluster de contrôle, ceci afin d'augmenter le débit de transfert des données entre le système et les Ptâches utilisateurs. En aucun cas cependant, pour les mêmes raisons évidentes de sécurité, un cluster mémoire ne devra avoir un pont en commun avec un cluster utilisateur.

La figure 5.3 donne un aperçu général de l'architecture du système ParObj. Nous expliquons plus en détail, dans les sections suivantes, le rôle des différents clusters et des différentes Ptâches du système.

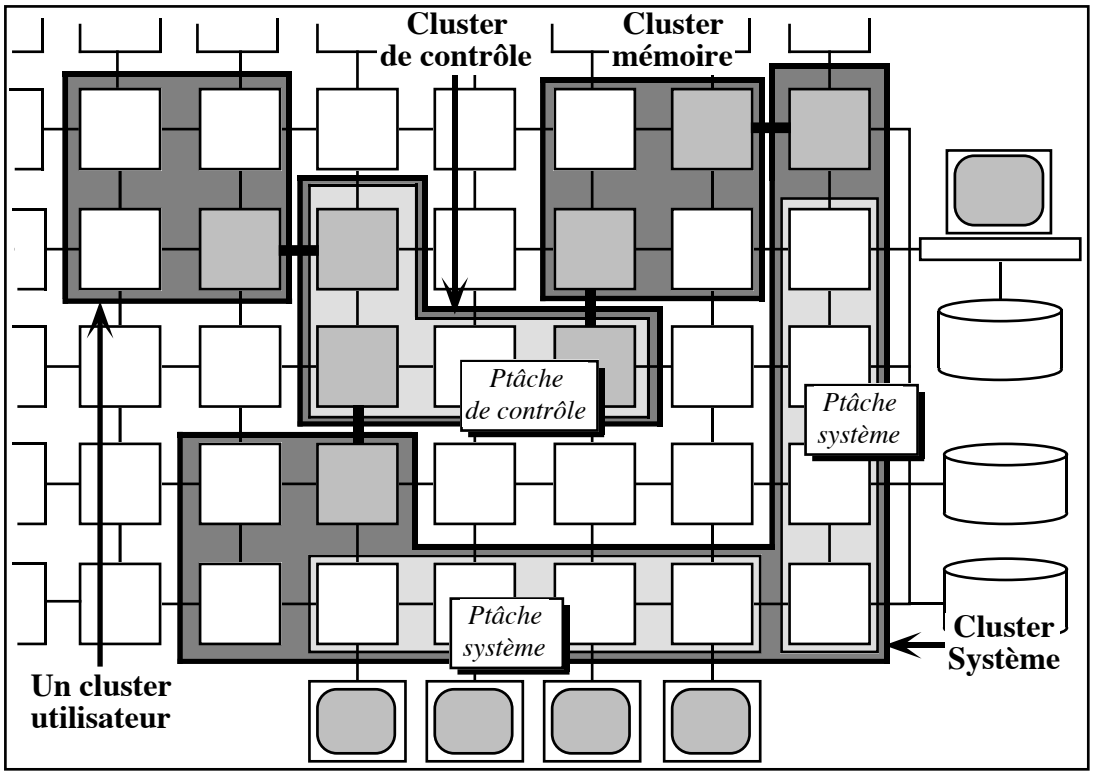


Figure 5.3. Architecture du sous-système ParObj

5.2.1.1 Le cluster de contrôle

Chaque fois qu'une Ptâche est lancée, le *chargeur* (qui est une Ptâche du cluster système) choisi un processeur de ce cluster. Ce processeur devient alors le *processeur de contrôle* de gestion de la Ptâche qui vient d'être lancée (Cf. chapitre **5.2.3 chargement d'une Ptâche** pour plus d'informations). Idéalement, à chaque création d'une Ptâche utilisateur, un processeur de contrôle devrait lui être assignée. Cependant, cette solution s'avére coûteuse en processeurs lorsque un nombre important de Ptâches s'exécutent sur la machine. Aussi, chaque processeur du cluster de contrôle devra être capable de gérer plusieurs Ptâches utilisateurs.

Un autre avantage de disposer d'un cluster de contrôle, réside dans le fait que la reconfiguration de ce cluster (par exemple à la suite de l'ajout de nouveaux processeurs) n'entraîne pas l'arrêt du cluster système, seules les communications entre les différentes Ptâches étant affectées.

5.2.1.2 Le cluster mémoire

Comme nous l'avons déjà expliqué dans le chapitre 5.1.2 sur la gestion mémoire, le rôle de la Ptâche mémoire est d'offrir une zone de mémoire auxiliaire aux applications. Ce service doit évidemment être complètement transparent à l'utilisateur.

5.2.1.3 Le cluster système

C'est le nerf du système, et il est composé de plusieurs Ptâches (Cf. figure 5.4) que nous appelons *Ptâches systèmes* par opposition aux Ptâches utilisateurs.

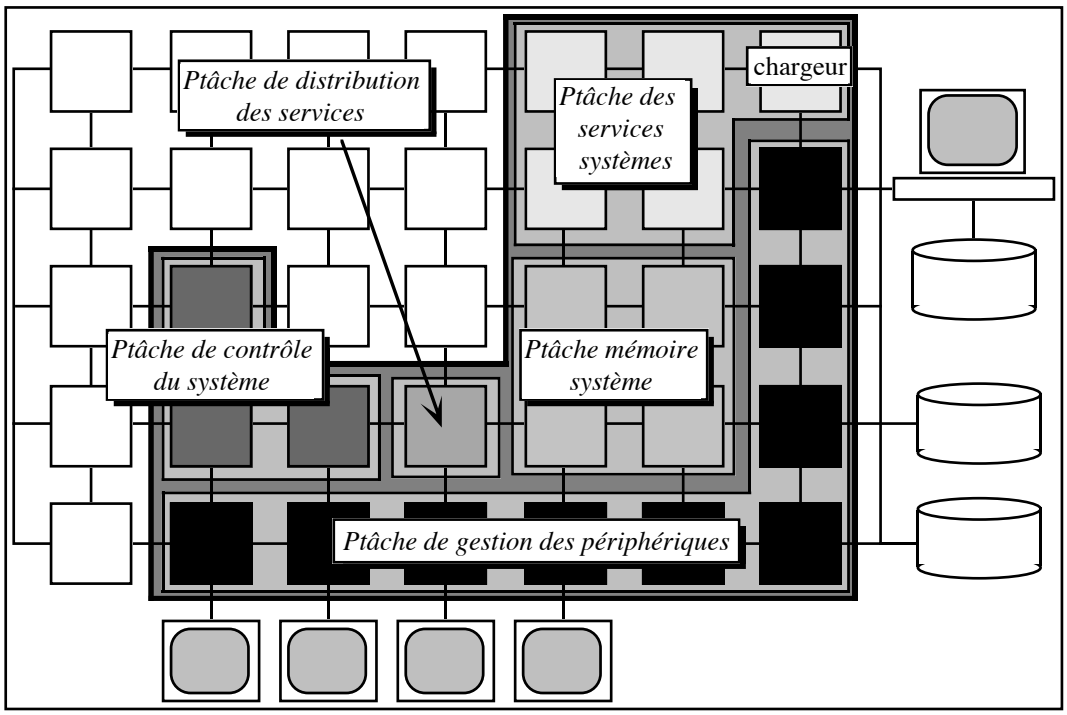


Figure 5.4. Le cluster système

Décrivons maintenant un peu plus en détail le contenu de chaque Ptâche. Dans le cluster système nous avons :

- la *Ptâche de contrôle du système* qui enregistre la localisation et l'état (suspendu, en cours d'exécution, etc.) des différentes Ptâches de la machine, qui garde la trace des différents environnements et des différentes périphériques disponibles (disques, écrans, imprimantes, etc.), et qui tient à jour l'état des ressources disponibles (processeurs, fichiers, ports, etc.).

- la *Ptâche des services système* qui contient des programmes systèmes tels que le *chargeur d'application*, l'*allocateur d'entités aux processeur*s, le *gestionnaire de fichiers*, etc.

- la *Ptâche de pilotage des périphériques* qui comme son nom l'indique s'occupe de la gestion des périphériques.

- la *Ptâche de reconfiguration de la machine* qui s'occupe de tout l'aspect reconfiguration de la machine (quand cette fonctionnalité est disponible). Ainsi, à titre d'exemple, sur les machines de type Supernode, cette Ptâche englobera tout les transputers qui gèrent les commutateurs du deuxième niveau.

- la *Ptâche mémoire système* qui sert de zone mémoire tampon pour les différents Ptâches du cluster système (par exemple pour stocker temporairement une Ptâche en cours de chargement).

- la *Ptâche de distribution de services* qui sert uniquement à recevoir les requêtes venant des Ptâches du "monde extérieur" (i.e. qui ne sont pas sur le cluster système) et à les rediriger vers les services correspondants (serveur de fichier, gestionnaire des Ptâches, etc.). Cette Ptâche est très utile lorsqu'il n'est plus possible d'atteindre directement un service système donné, par exemple parce que tous ces accès directs sont utilisés.

5.2.1.4 Les services système

Il nous reste à préciser le contenu de chaque processeur d'un cluster dans PARX. Nous avons vu que tout processeur dispose d'un µ-noyau qui offre différents niveaux de machines virtuelles. Au dessus de ce noyau, s'exécutent un certain nombre de services système (des serveurs) qui assurent la désignation, la gestion des entités, la gestion mémoire, etc. Cette approche serveur qui permet de bien structurer un système et de le rendre facilement extensible, est utilisée dans la plupart des systèmes distribués actuels.

Le *serveur de noms* se charge de générer les identificateurs d'entités, le *serveur d'entités* supervise la création et la destruction des entités. Il se charge aussi de résoudre les accès aux entités, et les problèmes de localisation et de migration de ces mêmes entités, Cf. chapitre **5.2.1 désignation et 5.2.4 gestion des entités à l'exécution** pour plus d'information. Le *serveur mémoire*, enfin, a pour responsabilité la gestion mémoire au sein de la Ptâche.

Tous ces serveurs fonctionnent de manière autonome lorsque l'information demandée peut être satisfaite localement (par exemple pour la création d'une entité statique), et s'adresse à des serveurs qui ont une vision plus globale dans le cas contraire. Ainsi la création d'une tâche globale va obliger le serveur de nom local à demander au serveur de nom du processeur de contrôle un identificateur pour cette tâche. De même si un serveur du processeur de contrôle ne peut découvrir l'information recherchée, il s'adressera à un serveur du cluster système. Si le cluster système ne peut satisfaire la requête, c'est une erreur.

La figure 5.5 détaille le cheminement d'une requête adressée au serveur d'entités, qui ne pouvant pas être satisfaite localement, part vers le serveur d'entités du processeur de contrôle. Celui-ci n'étant pas plus capable de traiter la requête, envoie finalement le message vers le cluster système, où le serveur d'entités système pourra normalement répondre à la demande.

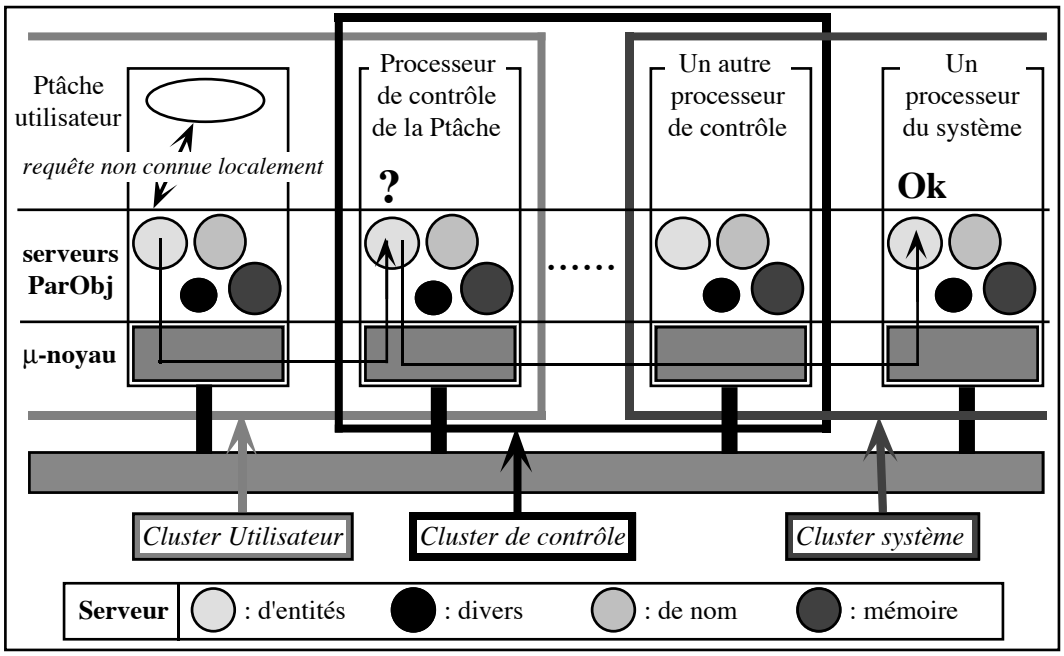


Figure 5.5. Communication entre les différents serveurs dans PARX

5.2.2 Désignation des entités

5.2.2.1 Identificateur

Lorsque le système crée une entité, il lui assigne un unique *identificateur d'entité* (**EID**). L'unicité de cet identificateur est fondamentale. En effet, si l'identificateur n'est pas unique, nous pouvons très bien imaginer des situations dans lesquelles deux entités `Ep11` et `Ep21` se retrouvent avec le même EID, par exemple `<241>`, sur des processeurs différents (Cf. partie gauche de la figure 5.6). Si `Ep11`, pour des raisons d'équilibrage de charge, est migrée sur le même processeur que `Ep21`, alors nous nous retrouvons avec un conflit d'accès, puisque le système ne saura pas à qui adresser les requêtes destinées à l'entité dont l'identificateur est `<241>` (Cf. partie droite de la figure 5.6).

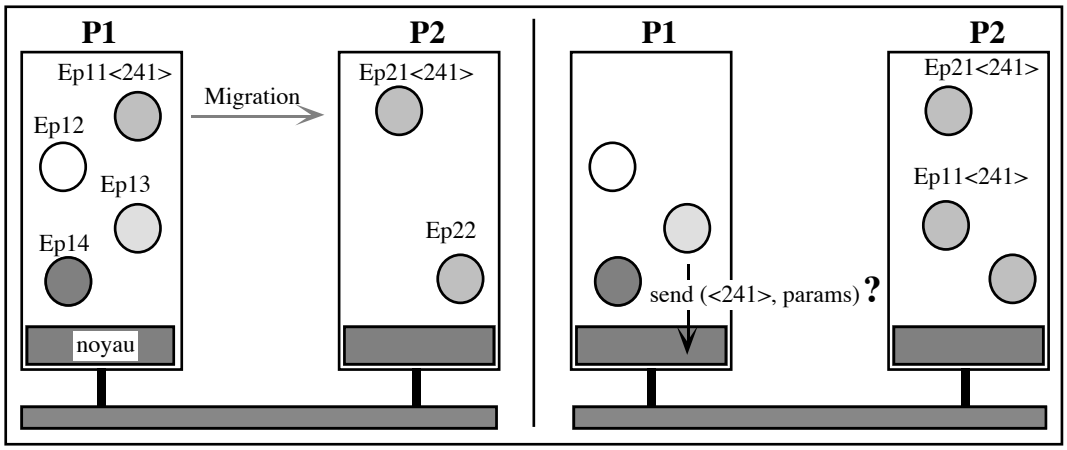


Figure 5.6. Conflit d'identificateur

Plusieurs solutions ont été envisagées pour la construction d'un tel identificateur. Par exemple, la solution qui consiste à utiliser un compteur unique que l'on incrémente pour chaque entité créée, si elle a le mérite d'être simple, présente un certain nombre d'inconvénients qui sont :

- plus le système est grand, et plus la granularité des objets reconnus par lui sera fine, alors plus la taille allouée aux EIDs devra être grande [BBK91], ce qui entraîne un accroissement de la taille des messages échangés entre entités,

- l'obligation de gérer le compteur d'une manière centralisée, ce qui peut amener des problèmes de conflits d'accès (si plusieurs entités sont créées en même temps, sur des processeurs différents, les requêtes vont arrivées au "même moment" sur le processeur qui gère le compteur) et des temps de réponses accrus (puisque les accès au compteur seront sérialisés).

- le mécanisme de translation qui est nécessaire pour résoudre un nom logique (l'EID) en une référence physique (processeur où l'entité réside, <adresse>) est complexe et coûteux, puisque l'EID ne contient aucune *indication de localisation.*

Nous proposons donc dans ce paragraphe, une manière de construire un mécanisme de désignation efficace et qui résout la plupart des problèmes précités.

La première chose à remarquer est que les entités gérées par ParObj suivent en fait une certaine hiérarchie : au plus haut de l'arbre nous trouvons le système, puis le cluster (pour la gestion des Ptâches) et le bunch (pour les programmes parallèles contrôlés par l'utilisateur). Sur chaque cluster, nous pouvons trouver un certain nombre de Ptâches qui s'exécutent en parallèle. Chaque Ptâche, quant à elle, étant constituée de plusieurs tâches, ports et objets. Et ainsi de suite… La figure 5.7 montre cette hiérarchie.

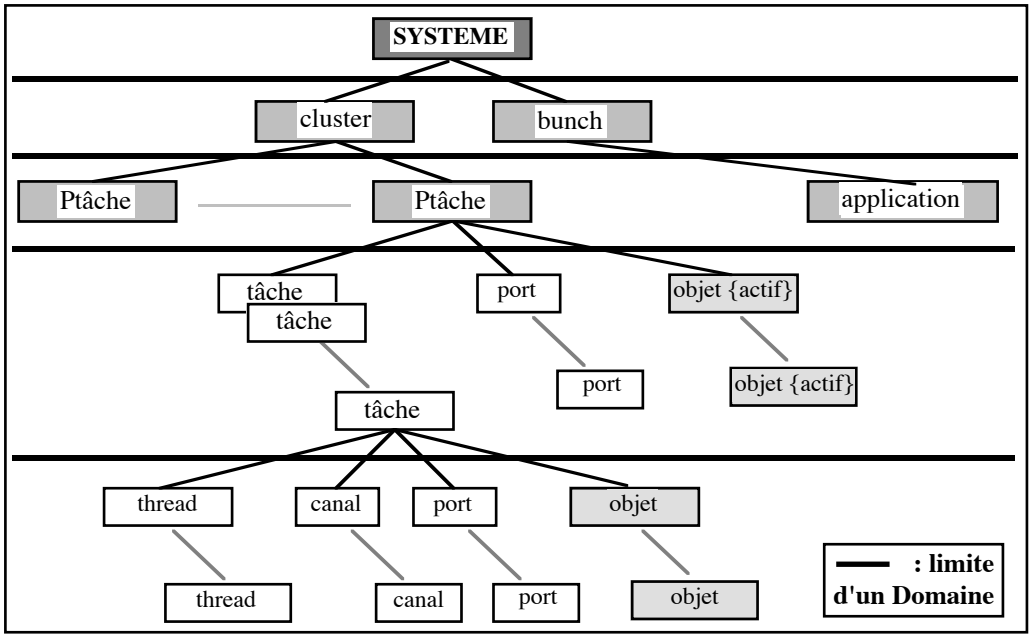
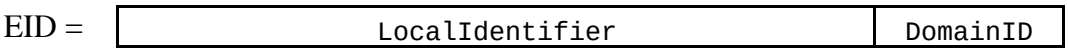


Figure 5.7. Hiérarchie des entités

Ainsi chaque entité n'a un sens qu'à l'intérieur d'un espace bien défini que nous appelons un **domaine**. Par exemple, une Ptâche ne peut être créée qu'à l'intérieur d'un Domaine Cluster. Cette notion de domaine, qui définit une sorte d'espace d'adressage structuré dans lequel les entités ont une signification, est utilisé dans un certain nombre de système, comme par exemple le SDO PEACE [Sc91].

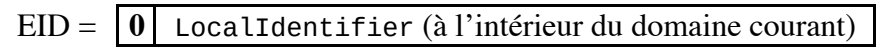
De cette notion, nous pouvons déduire qu'il est possible de découper un EID en deux parties, un champ pour le domaine dans lequel l'entité est créée, et un pour l'identificateur à l'intérieur du domaine :

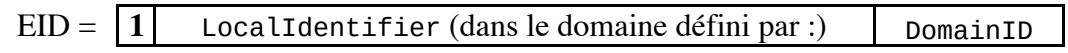


Ceci permet déjà de retrouver plus facilement une entité, puisque le système peut d'abord faire une recherche suivant l'identificateur du domaine.

D'autre part, nous avons introduit, dans le chapitre précédent, la notion d'entité privée ou publique. Ainsi, une entité privée n'est visible qu'à l'intérieur d'une Ptâche, ce qui signifie que le champ **`DomainID`** (en l'occurrence un identificateur de Ptâche dans ce cas) est non seulement superfétatoire, mais aussi coûteux, puisqu'à chaque communication, il faudra aussi le transmettre.

Conclusion, le champ `DomainID` n'a pas d'utilité à l'intérieur du domaine qui l'engendre. l'EID sera finalement codé de deux manières différentes, le bit de poids fort de l'EID permettant de les différentier (un 0 indique que l'entité est privée, un 1 qu'elle est publique) :

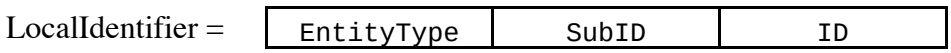




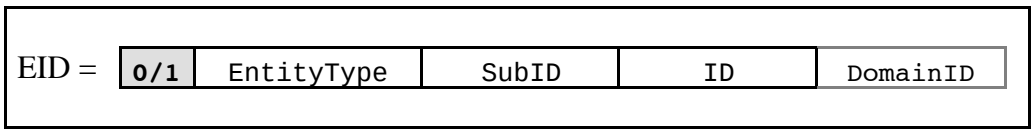
Il nous reste maintenant à décrire plus en détail le contenu de **`LocalIdentifier`**. Le but, ici aussi, est de simplifier l'attribution des identificateurs (en utilisant au maximum des identificateurs locaux à un processeur) et de pouvoir stocker certaines indications de localisation (comme c'était le cas avec le domaine), afin de simplifier le mécanisme de conversion nom logique/référence physique. Par exemple, le fait de savoir qu'un EID est défini par <champ1> <champ2>, permet de rechercher d'abord l'entité dans la table qui contient des <champ1>, puis dans la table qui contient des <champ2>.

Nous découpons donc le champ `LocalIdentifier` en trois parties : **`EntityType`**, **`SubID`** et **`ID`**. Le champ `EntityType` contient le type de l'entité i.e. cluster, Ptâche, tâche, port, objet, etc. La champ `ID` renferme l'identificateur (le nom) de l'entité qui est un des fils (dans la hiérarchie des entités) du domaine défini par `DomainID` (par exemple une tâche est l'un des fils d'une Ptâche). La champ `SubID` enfin, lorsqu'il est utilisé, renferme un identificateur local au processeur (un numéro de port, un numéro de

fragment, etc.) d'un des fils de l'entité identifiée par le champ `ID` (par exemple un thread représente un des fils d'une tâche). Ce qui donne :

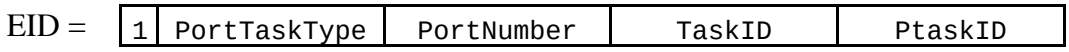


En résumé, la structure d'un EID se présente de la manière suivante :

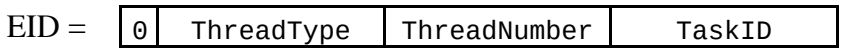


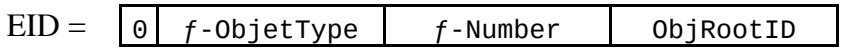
Ainsi, l'EID d'un cluster (entité publique avec le système comme domaine), et d'une Ptâche (entité publique avec un cluster comme domaine), seront respectivement codés :

De même, l'EID d'une tâche publique (dont le domaine est la Ptâche), et d'un port publique créé par une tâche (dont le domaine est aussi la Ptâche) seront respectivement codés :



Enfin, l'EID d'un thread (qui est une entité privée), et d'une *f*-objet privée seront respectivement codés (le domaine est la Ptâche, et les fils sont respectivement la tâche qui contient le thread et la racine de l'objet fractionné qui contrôle le fragment) :





5.2.2.2 Localisation

Bien que la structure d'un EID contienne des informations de localisations qui permette de retrouver plus facilement une entité (grâce au domaine en particulier), nous n'avons pas voulu, comme dans COOL ou PEACE par exemple, encoder directement la localisation physique de l'entité dans son identificateur. En effet, ParObj supportant la migration d'entité, nous voulions éviter l'utilisation de liens de poursuite (cette technique devenant très pénalisante lors de migration fréquente d'entités).

Comme, de plus, nous ne voulions pas utiliser de techniques basées sur de la diffusion (lorsqu'une entité ne peut être localisée) afin d'éviter de surcharger inutilement le ré-

seau de processeurs, nous avons décider d'utiliser un mécanisme basé sur des serveurs de désignation qui conservent la localisation des entités globales de la machine. Cependant, pour éviter le goulot d'étranglement d'une gestion centralisée, nous avons aussi décidé de créer sur chaque processeur, des serveurs de désignation qui font office de cache, en conservant localement les localisations d'entités distantes (ce qui permet de ne plus s'addresser au serveur centralisé). Le **Chapitre 6** décrit de façon détaillée l'implantation du mécanisme de désignation.

5.2.2.3 Droits d'accès (protection)

Afin d'assurer la protection des entités de ParObj, nous distinguons trois types de droits d'accès de base : les *accès en lecture* (sur un port, un objet, un cluster, etc.), les *accès en écriture* (sur un port, etc.) qui incluent aussi les opérations de type création et destruction d'entités, et les *accès pour exécution* (d'une Ptâche, d'une tâche, d'un thread, ou d'un objet actif) qui incluent aussi les opérations de gestion d'entités (Ptâche en particulier) et de remplacement de méthodes d'un objet.

Ces trois droits d'accès (rwx) ressemblent aux droits d'accès d'Unix mais possèdent une sémantique plus étendue dans la mesure où ils renferment plus de signification et sont interprétés selon le type de l'entité qui les possèdent. Ainsi par exemple, une Ptâche avec les droits rwx peut non seulement créer, modifier et exécuter les entités qui la compose, mais aussi les gérer.

De plus, une entité appartient toujours à un *propriétaire* (qui peut être une tâche pour un port, une Ptâche pour une tâche ou un objet, le système pour un cluster, etc.). Le propriétaire a généralement les droits rwx car il doit pouvoir par exemple détruire les entités qu'il a créées.

Il est parfois aussi nécessaire de disposer d'un droit au dessus de tous les autres, plus puissant même que le propriétaire, pour par exemple détruire une tâche qui ne nous appartient pas ou bien encore pour permettre à un déboggeur de manipuler des entités qui ne lui appartiennent pas. Sous Unix par exemple, l'utilisateur "*root*" (le super utilisateur) possède le maximum de droits possibles. Par analogie nous appellerons *droit d'accès supérieur* un tel droit.

Enfin, comme il est possible d'avoir simultanément de nombreux environnements (i.e. des sous-systèmes) qui s'exécutent sur des clusters différents, nous pensons qu'il est important d'introduire la notion de *droits d'accès pour un environnement donné* (comme PARX, PCTE, etc.). Les droits d'accès de l'entité contiennent donc un liste (un masque) des sous-systèmes pour lesquels l'entité a un sens. Par exemple, une entité créée dans l'environnement POSIX, est aussi valable dans un environnement Unix. Le contraire n'est malheureusement pas vrai.

Notons aussi qu'un utilisateur (super ou propriétaire) d'un environnement donné n'a aucun droit sur les entités d'un autre environnement.

En résumé, la forme la plus générale des *droits d'accès d'une entité* (**EAR**) peut être représentée de la manière suivante :

EAR =

Super	Owner	Read	Write	Execute	Environment

5.2.2.4 Capacités

Nous avons vu que toutes les entités dans le noyau PARX étaient protégées par des *capacités*. Une capacité est une structure généralement cryptée qui contient deux ou trois champs : une clé éventuelle (qui permet le codage/décodage de la capacité, et qui peut contenir d'autres informations utiles), un identificateur, et des droits d'accès décrivant le type d'opérations qui peuvent être réalisées sur cette entité.

Dans ParObj, les capacités ont le format suivant :

capacité =

EID	EAR	Key

5.2.3 Chargement d'une application parallèle

5.2.3.1 Format objet d'une application parallèle

Une Ptâche assure la projection en mémoire d'une application parallèle. Comme une Ptâche peut être constituée d'un certain nombres d'entités complètement autonomes, le format objet d'une application (format "a.out" sous Unix) sera constitué d'un certain nombre de sections, chaque section décrivant entièrement une entité. Chaque section étant à son tour découpée en un ensemble de sous-sections qui décrivent la structure de l'entité. Ainsi une tâche aura une section pour décrire le code de ses threads, une section pour les données, une pour les symboles, une éventuelle pour les informations de relocation, etc. Un objet aura des sections pour ses méthodes, ses données, et ses dépendances éventuelles avec d'autres entités (cas d'un objet fractionné).

D'autre part, afin d'aider le système dans son placement des différentes entités au moment du chargement, le format objet contiendra aussi un certain nombre d'informations pour leur allocation (comme les graphes des coûts de communications entre les entités, les temps d'exécution des tâches, le nombre minimal de processeurs au dessous duquel l'application ne peut être lancée, etc.). Enfin, le format objet sera aussi constitué de sections contenant des informations de relocation globale à la Ptâche, des symboles pour le déboggage, etc. La figure 5.8 détaille un tel format.

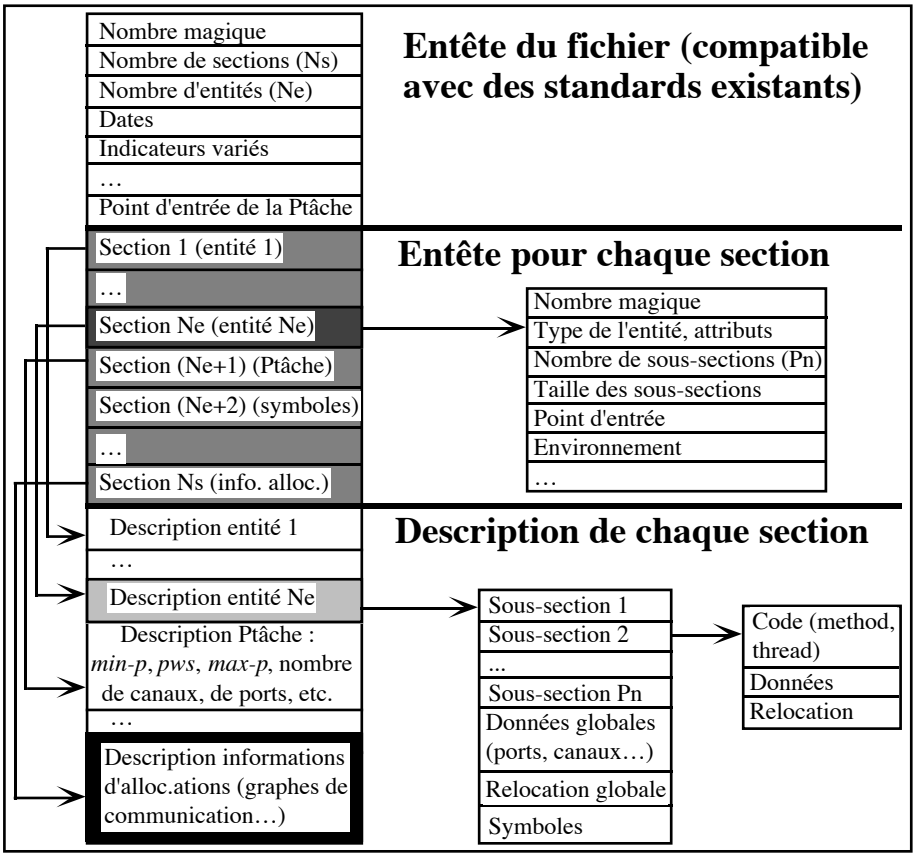


Figure 5.8. Un format objet pour une application parallèle

Ce format pourra aussi bien être dérivé du format TCOFF existant d'INMOS [SDPW91] s'il permet facilement le rajout d'extensions, ou de tout autre format (standard COFF par exemple) susceptible de répondre à nos besoins.

L'intérêt de cette représentation réside dans sa grande souplesse : en effet, il permet de rester relativement compatible avec les formats objets existants (grâce à l'organisation en sections, le chargeur peut par exemple complètement ignorer les informations de relocation, si le système n'est pas capable de les utiliser) et il autorise les extensions sans aucune modification du format existant (il suffit de rajouter une section avec une entête spécifique).

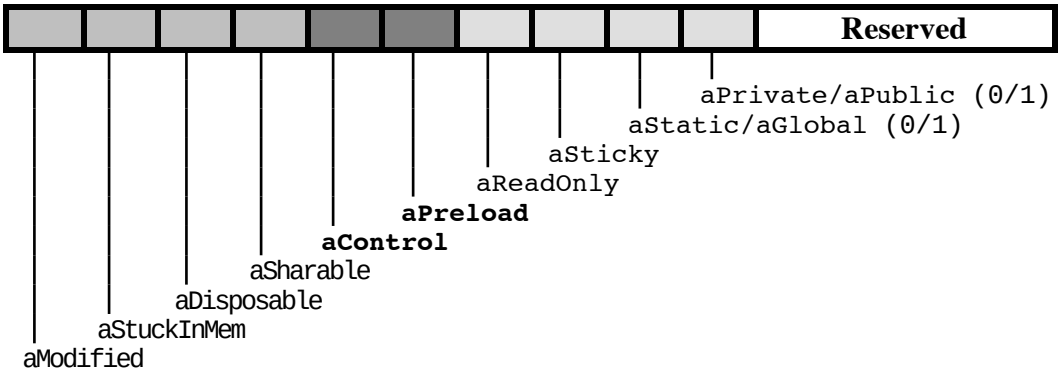
5.2.3.2 De nouveaux attributs pour les entités

Dans l'entête de chaque section, il existe un champ qui décrit les *attributs d'une entité*. Ces attributs indiquent au système comment l'entité doit être chargée, placée, gérée, etc. Nous connaissons déjà les attributs qui décrivent la visibilité d'une entité (privée ou publique) et sa référence (statique ou global, adhérent), Cf. **Chapitre 4,.2.2 Visibilité et 4.2.3 Référence** pour plus d'informations. Ainsi, les attributs statique, global et public détermine la structure de l'identificateur de l'entité.

Mais, nous avons aussi introduit d'autres concepts comme le cluster mémoire, et le cluster de contrôle. En particulier, nous avons vu qu'aller chercher une entité sur disque

pouvait être une opération coûteuse en temps, surtout si la machine est chargée. Aussi, par exemple, pouvoir spécifier que le programme doit être complètement placé en mémoire avant sont exécution, peut être une fonctionnalité intéressante. De même, une Ptâche peut avoir envie de placer certaines de ces entités sur le processeur de contrôle (qui, nous le verrons bientôt conserve la localisation de toute les entités globales de la Ptâche), afin par exemple, de faciliter des communications ou de la migration d'entités avec d'autres Ptâches.

Ces attributs sont stockés parmi les nombreux indicateurs que nous trouvons dans l'entête de chaque section décrivant une entité. Chaque attribut est codé sur un bit, un 1 indiquant sa présence et un 0 son absence. La structure du champ attribut est la suivante :



Détaillons maintenant un peu plus la signification de ces différents attributs. Les attributs *en italique* décrivent la visibilité et la référence d'une entité :

- **`aPrivate`**. Pour toute entité qui n'est visible qu'à l'intérieur d'une Ptâche. Une entité qui n'est pas privée est publique (**`aPublic`**).

- **`aStatic`**. Pour toute entité qui n'est visible que par les entités situées sur le même processeur et à condition que la liaison entre ces entités soit connue à la compilation. Une entité qui n'est pas statique est globale (**`aGlobal`**).

- **`aSticky`**. Pour tout entité qui ne doit pas être migrée (entité adhérente).

- **`aReadOnly`**. Pour toute entité dont l'accès est limité à la lecture (i.e. l'état de l'objet ne peut pas être modifié). C'est l'équivalent de l'attribut *"immuable"* des objets dans Emerald [BHLC87].

Les attributs **en gras** ont un rapport avec la phase de chargement :

- **`aPreload`**. Pour toute entité qui doit être chargée en mémoire avant l'exécution de la Ptâche, même si l'entité n'est pas immédiatement utilisée. Ce champ est par exemple très utile lorsque nous voulons faire des tests de performances pures, et que nous ne voulons pas prendre en compte le surcoût introduit par le transfert du disque vers la mémoire d'une entité créée dynamiquement. A l'opposé, si l'utilisateur sait que certaines entités de son programme ne seront utilisées que dans de

très rares occasions, il préférera ne pas les charger à l'avance, se réservant ainsi de la place pour les autres entités et évitant donc d'éventuels problèmes de manque de mémoire.

Cet attribut est très utilisé dans le système d'exploitation du Macintosh (Cf. [Apple88], *Resource Manager* pour plus d'informations) lors du chargement des ressources d'une application.

L'utilisateur dispose aussi d'un champ `aPreload` au niveau de la Ptâche. Si celui-ci est à un, alors toute l'application sera chargée en mémoire, et quelque soit la valeur de l'attribut `aPreload` de chaque entité. Si au contraire il est à zéro, alors chaque attribut sera pris en compte.

Ceci permet de réaliser ce que les concepteurs du SDO PEACE appellent "chargement incrémental" ou chargement à la demande. Par exemple, Un bibliothèque mathématique pourra avoir l'attribut `aPreload` de sa Ptâche à zéro. De cette manière, tant qu'elle n'est pas utilisée, seul le processeur de contrôle de cette Ptâche est initialisé. Lorqu'une Ptâche utilisateur fait appel à la bibliothèque, la tâche de contrôle du processeur de contrôle allouera alors un ou plusieurs processeurs et chargera tout ou une partie de la bibliothèque.

Note : Le champ `aPreload` d'un attribut est à 1 par défaut.

- **`aControl`**. Pour toute entité qui doit être placée sur le processeur de contrôle. Par exemple, toute tâche qui communique avec une tâche d'une autre Ptâche pourra être installée sur ce processeur, afin de réduire les coûts de communications, Cf. § sur la gestion des entités. A priori, seuls des utilisateurs privilégiés auront le droit d'utiliser cette fonctionnalité (pour des raisons évidentes de protection des accès entre Ptâches).

Enfin, les attributs restant servent pour la gestion mémoire de l'entité (les attributs `aDisposable`, `aStuckInMem`, et `aModified`) n'ayant un sens que lorque l'entité est effectivement présente en mémoire :

- **`aSharable`**. Pour toute entité qui peut être partageable. Par exemple, une entité en lecture seule (`aReadOnly`) peut être partageable.

- **`aDisposable`**. Indique au système que l'entité peut être définitivement retirée de la mémoire dans laquelle elle se trouve (par exemple à la suite de la destruction de l'entité). Cependant, cette entité n'est réellement purgée de la mémoire que lorsque le gestionnaire de mémoire à vraiment besoin de faire de la place.

  Cet attribut permet parfois de réduire les accès disque. En effet, supposons qu'une tâche charge en mémoire un objet en lecture seule, accède à ses données puis le détruise. Supposons maintenant, qu'une autre tâche ait besoin elle aussi d'utiliser cet objet. Alors, si l'objet a été irrémédiablement éliminé de la mémoire après sa destruction, le système devra de nouveau rechercher sur le disque l'objet

convoité. Si au contraire, il réside encore en mémoire, l'accès sera quasiment immédiat. Évidemment cela suppose qu'entre temps la mémoire n'ait pas été saturée.

- **`aStuckInMem`**. Pour toute entité qui ne doit pas être vidée sur disque même si le gestionnaire de mémoire a besoin de faire de la place. Cet attribut est très pratique pour éviter de vider sur disque des entités qui sont accédées très fréquemment, permettant d'éviter d'incessant va et vient entre les différents niveaux de swap. Attention, l'utilisation intempestive de cet attribut peut entraîner de graves problèmes de fonctionnement dans la gestion de l'application.

  Note : lorsque l'attribut `aStuckInMem` est à un, l'attribut `aDisposable` est complètement ignoré, et de plus, l'entité ne peut pas être migrée (i.e. elle se comporte comme si son champ `aSticky` était à un).

- **`aModified`**. Est positionné à un lorsqu'une entité a été modifiée. Seul le système peut le mettre à jour. Cet attribut est par exemple utilisé pour la gestion des objets persistants (si l'objet est détruit, et qu'il a été modifié, il faut répercuter les changements sur le disque).

La zone **`Reserved`** comportent des champs utilisés par le système et pour des extensions futures…

5.2.3.3 Chargement

Le rôle du chargeur consiste à prendre un format objet, à en extraire les différentes sections (tâches, objets, symboles éventuels), à les placer plus ou moins optimalement sur un cluster de processeurs, à indiquer à la Ptâche de contrôle du système qu'une nouvelle Ptâche à été créée, à récupérer du système un certain nombre d'informations (comme les identificateurs de ports des serveurs disque, imprimante, etc.), et à finalement lancer l'exécution de la Ptâche.

Nous nous plaçons évidemment dans la situation où l'application peut être lancée (c'est-à-dire qu'au moins `minp` ou `pws` processeurs sont disponibles dans le système).

Dans toute la suite de ce paragraphe, nous désignerons par **`Np`** le nombre de processeurs alloués à la Ptâche (ce nombre ne tient pas compte du processeur de contrôle et du nombre de processeurs mémoire qui ont été aussi alloués), et par **`Ne`** le nombre d'entités de la Ptâche qu'il faut charger en mémoire (i.e. celles dont l'attribut `aPreload` est à un).

Allocation

Une fois un cluster de `Np` processeurs (logiques) alloués, le chargeur doit récupérer les informations d'allocations qui sont stockées dans l'une des sections du format objet. Dans ce paragraphe, nous ne décrivons pas comment les algorithmes d'allocation statique d'entités aux processeurs fonctionnent, ce n'est pas le sujet de cette thèse. Cette matière ayant fait, et faisant toujours l'objet d'actives recherches au sein de l'équipe "SYMPA" [Eu90], [MT88], [Me88], [SuI89], [TM90], nous renvoyons le lecteur à ces travaux pour plus d'informations. Nous supposerons donc qu'il existe un ou plusieurs *allocateurs* disponibles dans la Ptâche des services système (nous utilisons volontairement le terme plusieurs, car il n'existe pas à l'heure actuelle d'algorithmes d'allocation efficaces pour n'importe quel type d'application).

Dans le cas où aucune information d'allocation n'est fournie, nous supposerons que la politique de placement du système est complètement aléatoire, bien qu'elle doive cependant respecter certaines contraintes telles que placer sur le processeur de contrôle les entités avec l'attribut `aControl`, et ne pas séparer les entités statiques des entités avec lesquelles elle communique. En cours d'exécution, le système pourra toujours essayer d'améliorer ce placement initial, grâce par exemple, à un algorithme d'équilibrage de charge.

La phase d'allocation est divisé en plusieurs étapes qui sont :

1. Le chargeur envoie à un allocateur toutes les informations sur la Ptâche dont il dispose (`Np`, `Ne`, graphe et coût de communication entre entités, temps d'exécution des tâches et objets actifs, nombre de services système accédés, etc.). Si un graphe particulier de processeurs physiques est réclamé (nous l'appellerons la *grappe prédéfinie* et nous le noterons **pß**), il est aussi envoyé à l'allocateur.

Note : ce dernier graphe n'est utile que sur des machines parallèles dont le mécanisme matériel ne permet pas toujours d'établir de multiples connexions entre processeurs (cas du Supernode). Alors fournir un graphe physique, permet d'indiquer au système que parmi les configurations possibles, nous préférons celle qui se rapproche le plus du graphe que nous fournissons. Enfin, si la machine n'est pas *dynamiquement reconfigurable* (i.e. les connexions entre processeurs sont fixées au démarrage de la machine), alors le graphe physique ne sert à rien.

2. L'allocateur retourne un graphe physique optimal (noté **oß**) à `Np` processeurs (avec leurs connexions) où le coût des communication entre entités est le plus faible, et le schéma de placement de ces entités (*graphe logique*) sur les différents processeurs du graphe oß. Si un pß était fourni, oß est égal à pß.

Nous supposerons dans la suite de l'algorithme qu'un graphe physique est nécessaire. Dans le cas contraire, il suffit d'ignorer les phases **3.** et **4.** de l'algorithme.

3. Le chargeur appelle alors le *topographe réseau* (un autre service système) dont le rôle est de vérifier la conformité du graphe oß avec l'architecture de la machine.

Ce programme retourne un graphe (**ß**) qui ressemble le plus possible au graphe oß. ß peut être le même que oß (surtout si c'est l'utilisateur qui l'avait fourni), mais il peut aussi être radicalement différent (considérez par exemple le cas où l'utilisateur a réclamé cinq accès directs vers le gestionnaire de fichiers et que seul un accès est disponible).

4. Si ß est différent de oß, le chargeur appelle de nouveau l'allocateur (avec ß comme paramètres). L'allocateur retourne alors le graphe de placement des Ne entités sur le graphe ß.

5. Le chargeur appelle alors le *configurateur de la machine* (qui fait partie de la Ptâche de reconfiguration de la machine) dont le rôle est d'établir les connexions physiques entre les différents processeurs de la grappe ß et de calculer les tables de routage qui doivent être placées sur chaque processeur de la grappe, afin que le routage des messages soit sans interblocage (Cf. [MMS91] pour plus d'informations).

Cette configuration est envoyée à la Ptâche de contrôle système pour qu'elle conserve la correspondance entre le cluster logique et la configuration physique. Le cluster de la Ptâche est donc maintenant configuré.

Le chargeur demande aussi à la Ptâche de contrôle système de créer la Ptâche utilisateur sur ce cluster. Bien que la Ptâche soit enregistrée, elle n'est toujours pas active et donc n'est pas encore visible par les autres Ptâches de la machine.

6. Enfin, le chargeur active d'une manière récursive le µ-noyau de chaque processeur (s'il était dormant), installe les différentes tables de routages, charge (s'ils n'étaient pas disponibles) et démarre les différents serveurs (*serveurs de noms*, *serveurs d'entités*, *serveur mémoire*, etc.) de ParObj.

Les Ne entités de la Ptâche peuvent maintenant enfin être chargées !

Placement

Le chargeur, grâce au graphe logique de placement des entités, installe alors chaque entité dans la mémoire du processeur correspondant, convertit l'entité en quelque chose de compréhensible pour le processeur si la machine parallèle est hétérogène, et réalise aussi les éventuelles relocations de code et de données des entités.

Une fois ces opérations effectuées, le chargeur exécute pour chaque entité, un certain nombre d'appels systèmes (par exemple `task_create()`, pour créer une tâche, puis `thread_create()` pour créer ses différentes Ptâches, `port_create()` pour créer un port, etc.) qui ont pour effet d'informer le serveur d'entité de chaque processeur qu'il doit allouer dans sa table une nouvelle structure appelée *bloc de contrôle d'entité* (**ECB**). Cette structure contient un certain nombre d'informations pour la gestion de l'entité et sera expliquée en détail dans le chapitre 5.2.4 suivant.

Nous avons vu dans le paragraphe "**Désignation des entités**" que le mécanisme de désignation de ParObj utilise des caches et des serveurs pour retrouver les entités. C'est pourquoi, le chargeur stocke sur le processeur de contrôle associé à la Ptâche, les EID et la localisation correspondante de toutes les entités globales (qui viennent d'être créeés) de la Ptâche. De plus, lorsque l'information est disponible (grâce par exemple au graphe entre entités), le chargeur indique aussi au serveur d'entités de chaque processeur, la localisation des entités distantes qui sont accédées par les entités situées sur ce processeur.

Le chargeur enfin, stocke dans une table (gérée par le serveur de nom) sur chaque processeur, le nom logique (qui est le nom par lequel l'utilisateur accédera à l'entité) des entités qui ont été préchargées (attribut `aPreload` à un), mais qui ne sont pas référencées lorsque la Ptâche démarre.

La figure 5.9 détaille le placement des différentes entités (7 objets et 4 tâches dans l'exemple) d'une Ptâche. Le chargeur ayant eu la connaissance de certaines dépendances entre entités, a pu par exemple indiquer au serveur d'entités du processeur P1 que O4 se trouve sur P2. Il a fait de même avec les serveurs d'entités sur P2 et P3 (ainsi ce serveur connaît la localisation de T1 et de O1). Le processeur de contrôle enfin est informé de la localisation de toutes les entités globales de la Ptâche. C'est pourquoi T4 et O7 (qui sont statiques) ne figurent pas dans sa table.

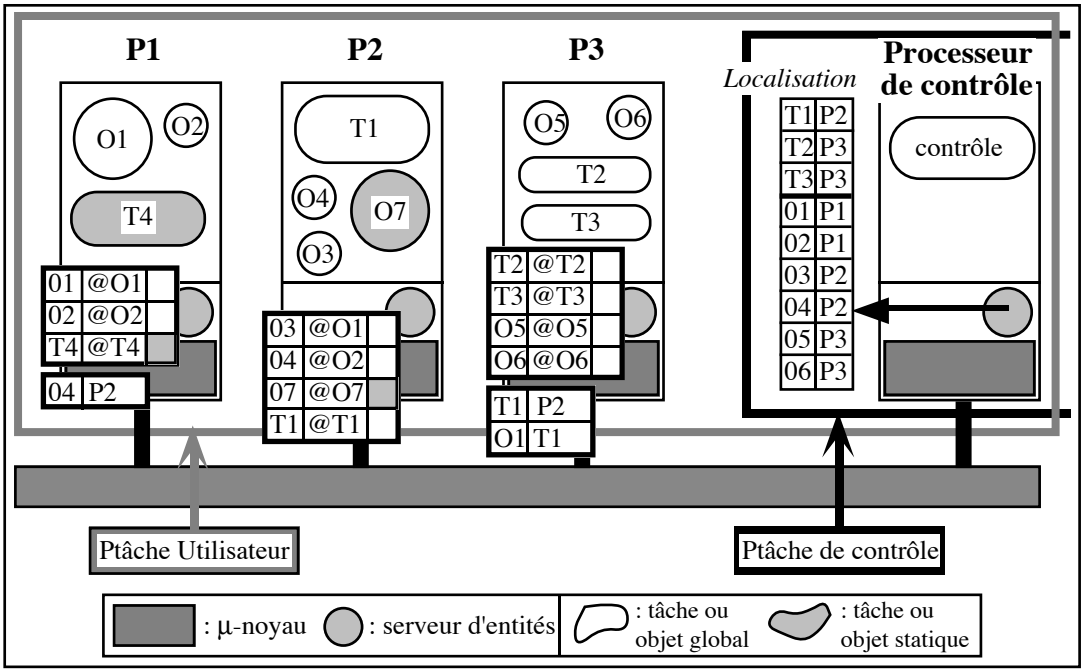


Figure 5.9. Placement des entités d'une Ptâche

Contrôle

Sur le processeur de contrôle associé à la Ptâche, le chargeur crée une *tâche de contrôle* dont le but est la gestion de la Ptâche et la synchronisation des différentes entités exécutables (tâches et objets actifs éventuels) qui la composent. Cette tâche a le maximum de droits possibles, par exemple elle peut détruire n'importe quelle entité de la Ptâche.

Dès que la phase de chargement est terminée, le chargeur démarre la tâche de contrôle, et indique au système que l'installation de la Ptâche s'est correctement passée. C'est alors le rôle de cette tâche de démarrer simultanément les entités exécutables, de les suspendre (par exemple à la suite d'une phase de reconfiguration du cluster de la Ptâche), de les redémarrer, etc., et d'indiquer au système (i.e. à la Ptâche de gestion système) les changements d'état de cette Ptâche.

5.2.4 Contrôle de l'exécution

Notre Ptâche a été chargée en mémoire, et la tâche de contrôle vient de la lancer. La Ptâche est située sur un cluster (qui est un domaine de communication et de protection) et dispose d'un processeur de contrôle dont le rôle est de contrôler la Ptâche, de garder la trace de la localisation et du statut des entités globales, de communiquer avec le cluster système et d'accéder éventuellement à un cluster mémoire.

Afin de comprendre comment les entités peuvent communiquer entre elles, et comment elles peuvent être localisées, nous devons maintenant décrire avec plus de précision le contenu des structures associées à chaque entité. Ces structures ont des noms différents suivant les systèmes (par exemple *descripteur d'accointances* [SGHM89], *descripteur d'objet* [Da90], [CALL89], *bloc de contrôle du processus* [DO91]) mais renferment généralement plus ou moins les mêmes type d'informations, à savoir :

- L'identificateur de l'entité (que nous avons appelé EID),
- les attributs de l'entité (global, privé, adhérent, etc.),
- la localisation de l'entité (adresse en mémoire, indice dans une table, etc.),
- et des statistiques sur l'entité (nombre de migrations [SGHM89], durée en mémoire, etc.)

5.2.4.1 Structures

Les informations pour la gestion des entités sont donc regroupées dans une structure que nous avons appelé *bloc de contrôle d'entité* (**ECB**), et peuvent être sous deux formats possibles.

Le premier format est utilisé par le serveur d'entités du processeur de contrôle pour garder la trace de toutes les entités globales, et par les serveurs d'entités locaux pour garder la trace d'une entité distante ("*remote*" en anglais d'où le 'R' dans le deuxième champ) :

ECB_r =

EID	R	eFlags	eLocalization	eNextEntry	eRefCount

le deuxième format, quant à lui, est utilisé pour gérer les entités c'est-à-dire celles qui se trouvent sur le même processeur que le serveur local d'entités :

ECB_l =

EID	L	eFlags	eLocalization	eNextEntry	eMoreEntry
eRefCount	eStatistics				

`EID` est l'identificateur de l'entité et a déjà longuement été détaillé. Lorsque l'entité est publique, nous rappelons que c'est la version longue de l'EID (avec l'identificateur du domaine) qui est stockée.

`eLocalization` contient la localisation complète de l'entité, que celle-ci soit distante ou bien locale. L'information de localisation est divisée en deux parties : la première contient le numéro de processeur sur lequel se trouve l'entité, la deuxième, qui permet de retrouver l'entité en mémoire, est variable et dépend du type de l'entité (pour un port ce sera son numéro dans la table des ports du µ-noyau, pour un objet ce sera l'adresse de sa table des méthodes, etc.).

`eFlags` contient les attributs de l'entité (`aStatic`, `aReadOnly`, …) et permet au système de vérifier rapidement si une requête est réalisable ou non (par exemple essayer de migrer une entité dont l'attribut `aSticky` est positionné à un, va lever une exception). En plus des attributs, ce champ renferme aussi le statut mémoire courant de l'entité. Les états possibles sont :

- **`sPreload`**. Pour les entités qui ont été préchargées en mémoire, mais qui n'ont pas encore été référencée.

- **`sInMemory`**. Pour les entités qui sont en mémoire et qui ont été référencées au moins une fois. Cela signifie que le serveur qui détient cette entité a un ECB_l valide.

- **`sNotAvailable`**. Utilisée uniquement sur le processeur de contrôle. Indique à un serveur d'entités local qui fait une requête de localisation, que l'entité correspondant n'est pas dans un état stable (en cours de migration, en cours de chargement, etc.). La requête est donc déposée dans une file d'attente (sur le processeur de contrôle), et la réponse sera envoyée uniquement lorsque le statut de l'entité aura changé.

- **`sNoMoreHere`**. Utilisé pour indiquer que l'entité ne se trouve plus sur ce processeur (soit parce que l'entité a été détruite[13], soit parce qu'elle a migré). Suivant le

[13] Nous rappelons qu'une entité détruite (attribut `aDisposable` positionné à 1) n'est pas immédiatement vidée de la mémoire, ceci afin d'optimiser le cas où l'entité est créée de nouveau quelques temps plus tard. C'est pourquoi son bloc de contrôle peut encore être présent dans le système.

mécanisme de migration implanté, il faudra donc essayer de retrouver l'entité. Par exemple en s'adresser au processeur de contrôle, en suivant un pointeur de relégation de l'information [BBK91], en diffusant [De92] le messags sur tous les processeurs du cluster, etc.

- **`sSwapped`**. Pour les entités qui ne sont plus en mémoire principale (par exemple parce qu'elles ont été vidées sur une zone de swap).

- **`s????`**. Si les valeurs précitées s'avèrent insuffisantes. En particulier, il est fort possible que suivant le type de gestion mémoire implanté d'autres états devront être introduits (par exemple dans le cas d'une gestion répartie des pages mémoire, quel statut doit-on attribuer à une entité dont certaines pages mémoire ne se trouvent pas sur le même processeur que son bloc de contrôle ?)

Afin de minimiser les temps de recherche, les blocs de contrôle sont chaînées entre eux selon des valeurs de *"hash-code"* qui dépendent de la localisation de l'entité. Le champ **`eNextEntry`** permet donc le parcours de chaque liste, à la recherche du bon ECB.

`eRefCount` est un *compteur de référence* qui permet de connaître le nombre d'accès à l'objet (par exemple, l'ECB d'un objet partagé par deux tâches aura un eRefCount à 2). Lorsque ce compteur tombe à zéro, cela signifie que l'entité n'est plus référencée, et donc l'ECB correspondant peut être supprimé. Lorsque l'entité est locale, `eRefCount` contient alors un "pointeur" (**`eMoreEntry`**) qui permet de retrouver les informations supplémentaires du bloc de contrôle (compteur de référence et statistiques).

`eStatistics` enfin contient des informations pour aider le système à prendre des décisions concernant la migration des entités et l'équilibrage de charge. Typiquement, ce champ devra contenir des statistiques sur la fréquence d'utilisation de l'entité, le volume d'informations échangées entre elle et les autres entités, et éventuellement d'autres données comme par exemple le nombre de migration déjà effectuée (SOS), ou bien l'age relatif de la donnée (Emerald), etc.

5.2.4.2 Création dynamique

La création dynamique d'entité est autorisée dans ParObj. Par défaut, une entité est créée sur le processeur où l'invocation a eu lieu. Cependant des situations existent pour lesquelles une création distante est souhaitable. Par exemple créer une entité sur un processeur qui n'a presque plus de mémoire disponible, peut obliger le gestionnaire de mémoire à vider sur une zone de swap d'autres entités. Ce qui risque de se traduire par une baisse générale des performances de l'application.

D'autre part, l'utilisateur peut avoir la connaissance d'un service sur un autre processeur et donc vouloir explicitement la création de son entité sur celui-ci, afin de diminuer le volume des informations échangées entre ces deux processeurs. Ce choix de placement à la création est par exemple utilisé dans le SDO Argus [LCJS87], [Li88].

La création elle même ne pose pas de problème particulier. Sur le processeur où est créée l'entité, le serveur d'entités associé alloue un nouveau bloc de contrôle (`ECBl`) avec l'identificateur de l'entité (obtenu par l'appel au serveur de noms), sa localisation sur le processeur, etc. Si l'objet est global, il informe aussi le serveur d'entités du processeur de contrôle pour que celui-ci alloue une entrée (`ECBr`) dans la table des entités globales connues par la Ptâche. Si de plus, l'entité est publique, il informe le serveur d'entités du processeur de contrôle système pour que celui-ci alloue une entrée (`ECBr`) dans la table des entités publiques connues par toutes les Ptâches de la machine.

5.2.4.3 Invocation d'une entité privée

Au début de l'exécution de la Ptâche, chaque serveur d'entités connaît les localisations des entités locales au processeur, et éventuellement celles sur des entités distantes (stockées dans un `ECBr`), si des renseignements sur leur localisation étaient disponibles.

Lorsqu'une entité invoque une entité globale[14] distante, un message (qui contient entre autres l'EID de l'entité cible) est d'abord envoyé au serveur local. Celui-ci consulte alors ses tables internes. S'il trouve l'`EID` spécifié, il renvoie dans la requête, la localisation de l'entité distance, ce qui permet à l'entité locale l'envoi de son message vers l'entité cible (Cf. figure 5.10).

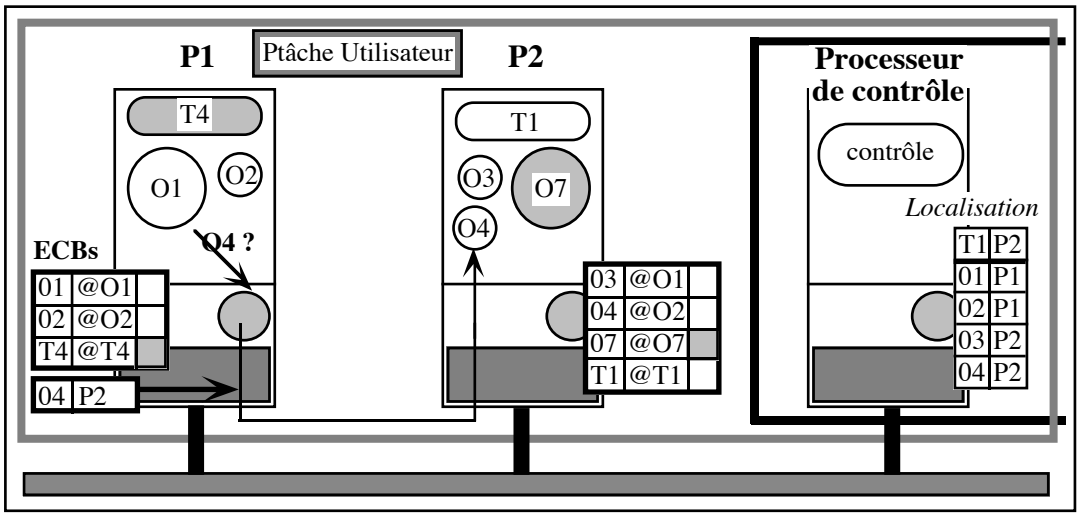


Figure 5.10. Invocation d'une entité distante connue du serveur local

Si le serveur local ne peut pas trouver la cible spécifiée, il envoie une *requête de localisation* vers le processeur de contrôle. Si le serveur d'entités du processeur de contrôle détient l'information, il renvoie un message au serveur local contenant la localisation de l'entité spécifiée. Le serveur local n'a plus qu'à créer un nouvel `ECBr` identifiant l'entité distante (Cf. figure 5.11). A partir de ce moment, plus aucune communication im-

[14] L'invocation d'une entité statique se fait directement par appel de procédures (puisque la liaison a pu être résolue à la compilation et que les deux entités se trouvent sur le même processeur) et ne passe donc pas par le système.

pliquant cette entité n'est nécessaire avec le processeur de contrôle, et toute requête la désignant sera automatiquement routée vers le bon processeur.

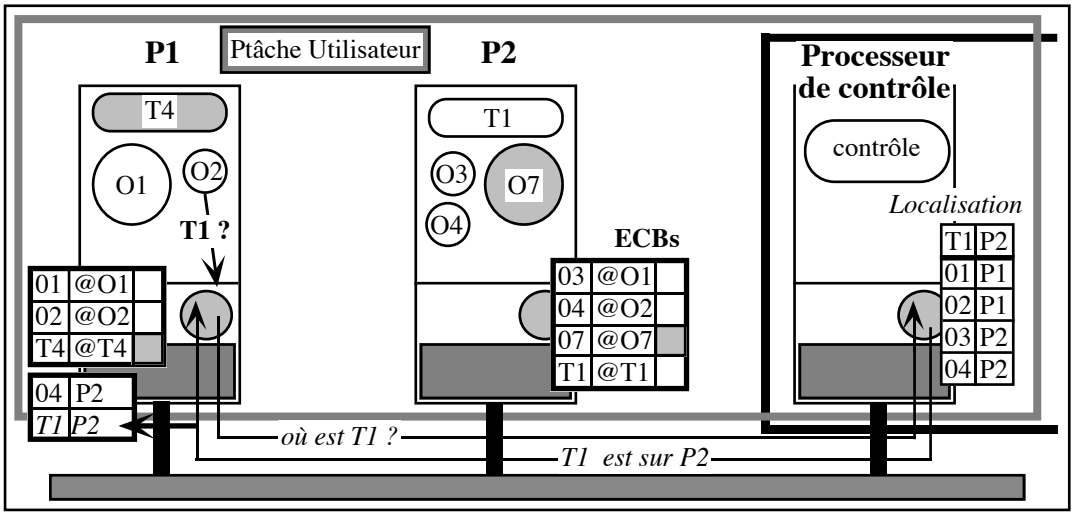


Figure 5.11. Invocation d'une entité distante non connue du serveur local

Si le serveur d'entité du processeur de contrôle ne peut pas trouver l'information, cela signifie que soit l'entité n'est pas dans la mémoire de la Ptâche (swappée si le champ `eStatus` vaut `sSwapped`, sur disque dans le cas contraire, et dans ce cas le serveur demande au système le chargement de l'entité, etc.), soit elle est temporairement non disponible (`eStatus` vaut `sNotAvailable`). Dans tous les cas, la requête est déposée dans une file d'attente, et le serveur de contrôle renvoie une réponse au serveur local uniquement lorsque le statut de l'entité devient `sInMemory`. Si le serveur ne peut retrouver l'entité c'est une erreur, et une exception est levée.

5.2.4.4 Invocation d'une entité sur un autre cluster

Nous avons vu que la communication entre clusters se faisait par l'intermédiaire des différents processeurs de contrôle du cluster de contrôle.

Lorsqu'une entité désire être accédée par des entités extérieures à sa Ptâche, elle doit être associée à un port public qui doit être déclaré comme rendant un service donné (grâce à l'appel système `port_publish(…)`). Lorsqu'une autre entité sur un processeur distant désire accéder à ce service, elle exécute un `port_lookup(…)` qui a pour conséquence de premièrement retourner une capacité associée à se service, et deuxièmement de créer un ECB_r avec dans le champ `eLocalisation`, le numéro de processeur distant et le numéro de port sur ce processeur.

Mais ce processeur n'appartenant pas au cluster de la Ptâche, son numéro n'apparaît pas dans les tables de routage de la Ptâche. Conclusion, lorsque un serveur d'entités local récupère la localisation de l'entité distante, il ne peut lui envoyer directement le message, puisqu'il ne connaît pas le chemin pour y aller. Dans ce cas, il envoie systématiquement le message vers le processeur de contrôle de la Ptâche (en passant par le pont de communication), en espérant que celui-ci saura correctement router le message.

Communication avec le cluster système

Les communications avec le cluster système pouvant être relativement fréquentes, il ne faut pas que les messages passent leur temps à rebondir de serveurs d'entités à serveurs d'entités, à la recherche du destinataire final. Or, lorsqu'un message est destiné au système, son chemin passe invariablement par les processeurs de contrôle de chaque Ptâche, puis par le ou les ponts entre le cluster de contrôle et le cluster système. Conclusion, ce mécanisme d'acheminement peut être directement pris en compte au niveau du protocole de routage.

Pour cela, à tout processeur du cluster système est assigné un numéro particulier qui permet au mécanisme de routage de le reconnaître aisément (par exemple en mettant à un le bit de poids fort du numéro de processeur). Ainsi, lorsque un tel numéro est rencontré, le message correspondant est automatiquement routé de pont de communication en pont de communication, sans passer par les serveurs d'entités. Lorsque le message arrive enfin dans le cluster système, il est délivré directement au bon endroit puisque le numéro de processeur destination est connu des tables de routage.

Note : ce procédé (le codage particulier du numéro de processeur) peut être appliqué à tous les clusters fournissant un service système. Par exemple, un cluster mémoire devra pouvoir lui aussi être accédé le plus rapidement possible.

Communication avec un cluster utilisateur

Après l'exécution de la procédure `port_lookup(…)` qui génère un message à destination de la Ptâche de contrôle du système (sur le cluster système)[15], le serveur d'entités local dispose d'un ECB_r qui contient l'`EID` et la localisation de l'entité qui se trouve sur un autre cluster.

Lorsque l'entité locale E_l veut communiquer avec une entité distante E_d, un message est envoyé au serveur d'entités local qui, ne sachant où envoyer le message, le route vers le serveur d'entités du processeur de contrôle de la Ptâche (que nous appelons `SEPctrl`). Si c'est la première fois qu'une requête est émise vers E_d, le serveur de contrôle ne sait pas plus où envoyer le message. Il va alors de nouveau s'adresser au cluster système et lui demander "*vers quel processeur de mon cluster dois-je envoyer le message si je veux communiquer avec* E_d *?*". Le système grâce à E_d retrouve la Ptâche associée, et donc le processeur de contrôle correspondant. Il peut donc renvoyer l'information demandée. Lorsqu'il reçoit une réponse, le `SEPctrl` crée un ECB_r qui va associer l'`EID` de E_d avec le numéro de processeur du processeur de contrôle de la Ptâche distante. Les messages suivants pourront alors être systématiquement routés vers ce processeur. Lorsque le message arrive sur le processeur de contrôle de la Ptâche distante, la tâche de contrôle qui connait la localisation de Ed, peut alors envoyer la reqête sur le bon processeur, Cf. Figure 5.12.

---

15 Ce message est évidemment routé en utilisant le protocole défini dans le paragraphe précédent.

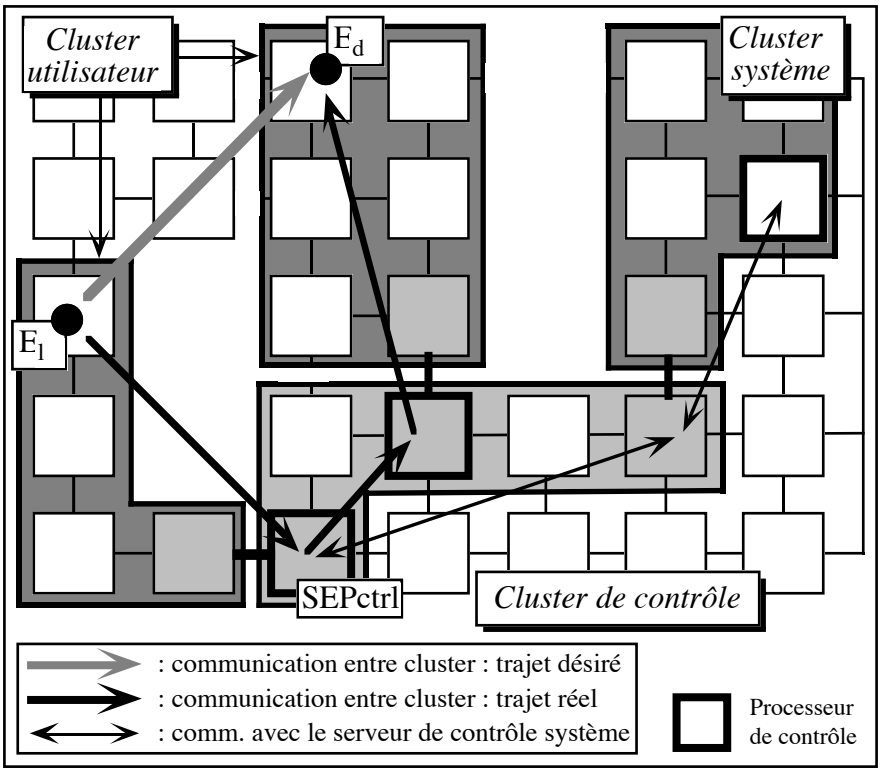


Figure 5.12. Communication entre deux Ptâches

Communication par un chemin direct

Afin d'accélérer la communication entre clusters, et si le matériel de la machine le permet, il devra être possible de réclamer (par exemple au moment du chargement) un ou plusieurs *chemins directs* vers un cluster donné. Ces chemins permettront d'éviter le passage par le cluster de contrôle, en autorisant une communication entre deux processeurs de deux clusters différents.

La figure 5.13 résume les différents chemins que prennent les messages lors de la communication entre clusters différents.

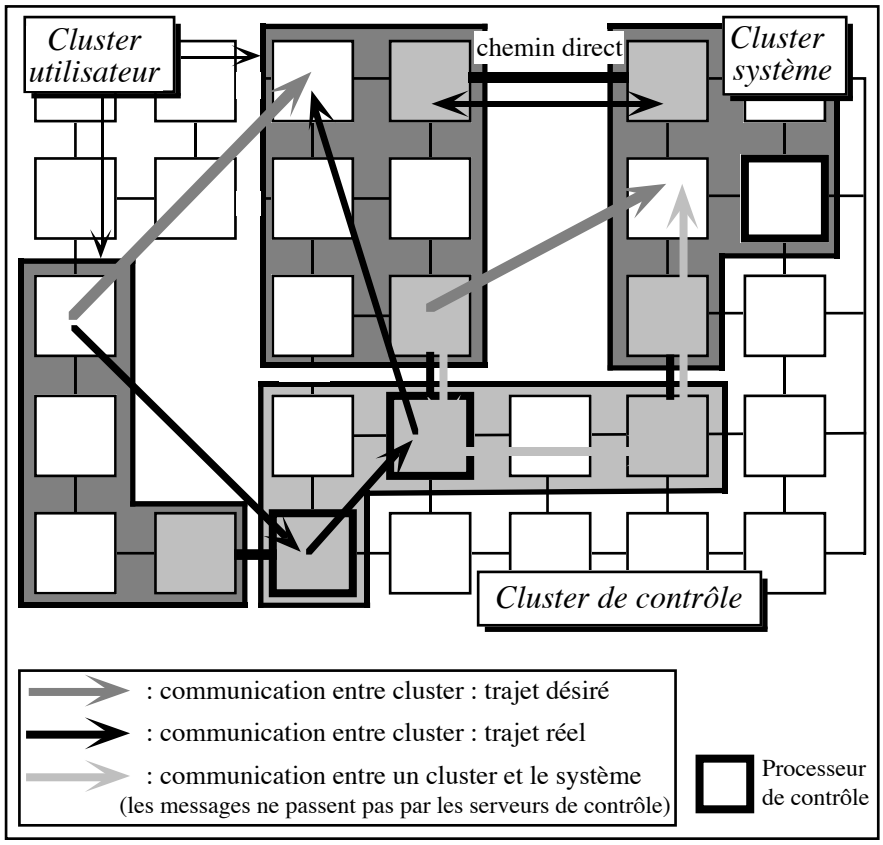


Figure 5.13. Les différents moyens de communication entre Ptâches

III. Mise en œuvre

Chapitre 6 : Réalisation

Le choix d'une réalisation pertinente et rapidement utilisable, lorsqu'il s'agit de logiciels systèmes est généralement problématique. Ainsi par exemple, un système gestionnaire de fichiers (parallèle), développé dans un coin, ne pourra vraiment être démontré et testé que si le programmeur dispose aussi d'un système de gestion mémoire approprié. C'est pourquoi il est souvent difficile de présenter des travaux pratiques, lorsque la partie sur laquelle nous travaillons, ne représente qu'une des briques de base d'un ensemble bien plus complexe – dans notre cas, un SDO (système distribués à objets).

Nous avons nous aussi été confronté au problème du choix. Fallait-il enrichir le noyau avec des objets, permettant par exemple de tester l'efficacité des scripts de synchronisation ? Malheureusement, sans gestion mémoire et sans moyen de désigner efficement ces objets, l'entreprise eut été vouée à l'échec.

En effet, dans la version de PARX que nous utilisons actuellement au sein de l'équipe, toute entité doit être désignée par un couple (*processeur, indice dans une table*). Ceci provoque souvent des erreurs de programmation, car il arrive fréquemment que l'on se trompe dans les indices à passer aux différents appels système.

Aussi, nous avons pensé que passer d'une désignation physique à une désignation logique permettrait de grandement simplifier l'utilisation du noyau, et autoriserait la construction des mécanismes de plus haut niveau tels que la gestion mémoire, ou la notion d'objet.

6.1 Implantation de la désignation

6.1.1 Structure d'un identificateur d'entité et d'une capacité

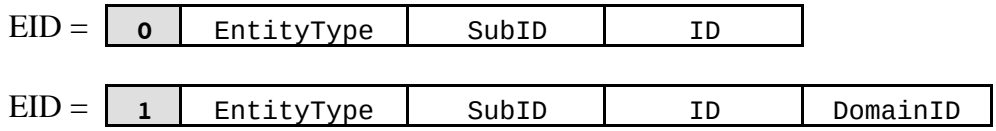
Nous avons détaillé dans le chapitre sur la **Gestion des entités dans ParObj**, le format d'un identificateur d'objet (EID) et de sa capacité associée. C'est pourquoi, nous décrivons brièvement les choix que nous avons pris dans la réalisation de la désignation.

Après l'étude d'un certain nombre de systèmes distribués basés sur des capacités (Accent, Amœba, Clouds, Eden, Helios, SOS, etc.), nous avons décidé de coder les capacités sur 64 bits (8 octets). Cette taille représente en effet un bon compromis entre

deux aspirations contradictoires : avoir une taille de capacité la plus petite possible pour minimiser le nombre d'octets tranférés entre processeurs, et pouvoir stocker toutes les informations nécessaires telles que l'identification de l'entité, les droits d'accès associés et une clé éventuelle pour le cryptage/décryptage de la capacité (afin de la rendre illisible à un utilisateur indiscret).

Afin de conserver la plus grande partie de la capacité pour la représentation de l'identificateur d'une entité, nous avons décidé de coder les droits d'accès et la clé sur 16 bits, ce qui nous laisse 48 bits pour coder l'EID. De par sa nature hiérarchique (Cf. schéma ci-dessous), nous avons constaté que chaque composant de l'EID n'a pas besoin d'être décrit sur beaucoup de bits.

Nous rappelons qu'un EID a respectivement le format suivant, selon que l'entité est privée ou publique (Cf. chapitre 5 pour plus d'informations) :

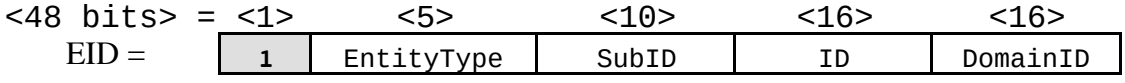


Comme nous voulions faire tenir la taille d'un EID privé sur 32 bits (car c'est généralement la taille d'un mot machine et par conséquent les accès sont très rapides), il nous restait 16 bits pour coder l'identificateur de domaine (`DomainID`). Pour rester homogène avec cette identificateur, nous avons aussi décider de coder l'`ID` sur la même taille.

Ceci permet alors, de créer un maximum de 65535 clusters (zéro n'étant jamais utilisée comme valeur d'identificateur) dans la machine, avec un maximum de 65535 Ptâches par cluster, et un maximum de 65535 tâches par Ptâche. Même dans le cas d'une machine comportant un très grand nombre de processeurs (10000 et plus), nous pouvons constater que ces valeurs limites seront rarement atteintes.

Comme le bit de poids fort de l'EID est utilisé pour coder l'état privé/public, nous disposons de 32 - 16 - 1 = 15 bits pour représenter le reste des informations relatives à l'entité. Aprés analyse du nombre d'entités gérés dans ParObj, nous avons finalement décidé d'utiliser 5 bits pour coder l'`EntityType` (ce qui permet de gérer 32 entités différentes) et 10 bits pour le `SubID` (ce qui permet par exemple d'avoir un maximum de 1023 threads, 1023 ports, 1023 canaux, et 1023 objets par tâche).

En résumé, un EID est codé de la manière suivante :



Les types correspondant utilisés sont (le noyau PARX étant écrit en C, nous utiliserons ce langage pour décrire les différentes structures de données et appels système) :

```
typedef unsigned short   t_id   ;          /* 16 bits */
typedef unsigned short   t_id10 ;          /* 10 bits */
typedef unsigned char    t_ar   ;          /*  8 bits */
typedef unsigned char    t_key  ;          /*  8 bits */

/*
 *  sVis vaut 0 (vPRIVATE) si l'entité est privée et 1 (vPUBLIC) sinon.
 */
typedef struct short_eid {                 /* 32 bits */
   unsigned    sVis      :  1 ;           /*  1 bits */
   unsigned    sType     :  5 ;           /*  5 bits */
   unsigned    sSubID    : 10 ;           /* 10 bits */
   unsigned    sID       : 16 ;           /* 16 bits */
} t_short_eid ;


typedef struct eid {                       /* 48 bits */
   t_short_eid  eShort  ;
   t_id         eDomainID ;
} t_eid ;


typedef struct capability {                /* 64 bits */
   t_short_eid  cShort         ;
   t_id         cDomainID      ;
   t_ar         cAccessRights  ;           /*  8 bits */
   t_key        cKey           ;           /*  8 bits */
} t_capability ;
```

Note : dans la version actuelle, les capacités ne sont pas cryptées.

6.1.2 Fonctionnement général du mécanisme de désignation

Tous les appels système que nous avons réalisés, et que nous appelerons *appels système de haut niveau*, ont le format suivant :

```
t_error system_call(&<capability>{,<other parameters>})
```

Par exemple l'envoi d'un message sur un port, se fait grâce à la fonction :

```
t_error port_send(&<capability>,<size>,<message>)
```

Par opposition, l'envoi de ce même message avec la version précédente du noyau, se faisait grâce à l'*appel système de bas niveau* suivant :

```
t_result asp_send(<proc>,<index>,<size>,<message>)
```

Comme nous avons vu qu'une capacité ne contient pas d'informations de localisation, nous devons trouver un moyen d'établir la correspondance entre une désignation logique (EID) et une désignation physique (`proc,indice`), puisque chaque appel système de haut niveau devra, tôt ou tard, appeler son homologue de bas niveau pour réaliser l'opération demandée.

La solution, déjà détaillée dans le chapitre 5 (**Gestion des entités dans ParObj**), réside dans l'utilisation de différents serveurs installés sur chaque processeur, ces serveurs communiquant éventuellement entre eux pour obtenir les informations qui leur manquent. Nous rappelons aussi qu'il existe, pour chaque Ptâche, un serveur particulier (le serveur de contrôle) dont le rôle est de conserver la trace de toutes les entités globales de la Ptâche.

Détaillons maintenant comment s'exécute un appel système de haut niveau. Son fonctionnement est schématisé dans la figure 6.1.

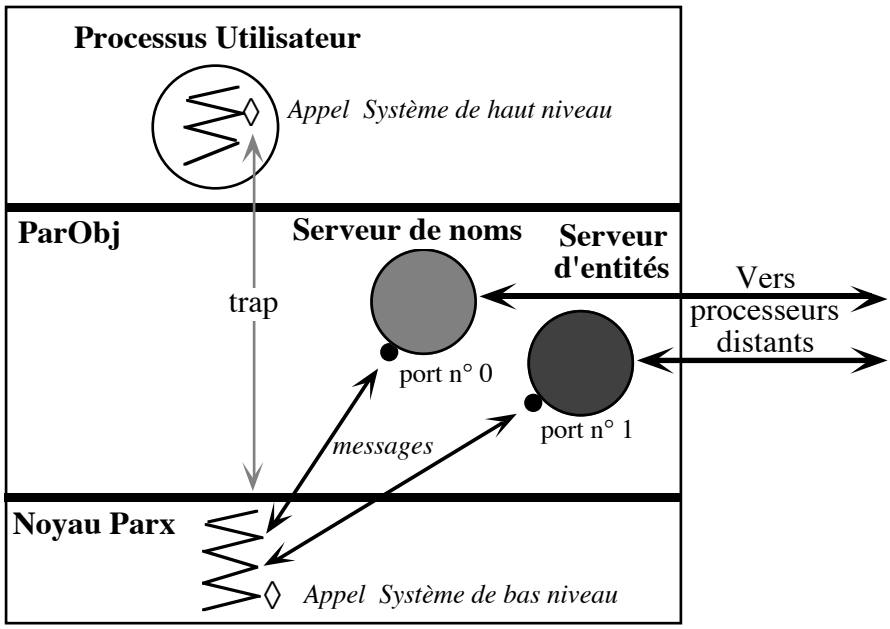


Figure 6.1. Fonctionnement d'un appel système de haut niveau

Après exécution d'un trap[16], nous nous retrouvons dans le noyau. Là, chaque routine associée, exécute un code qui se déroule selon le principe suivant :

(1) Récupération des paramètres

(2) Décryptage de ou des capacités

(3) Vérification des droits d'accès et contrôle de type

Par exemple lors d'un chan_out, il faut vérifier que le droit d'accès WRITE est présent, et que le type de l'EID correspond bien à un canal.

(4) Construction et envoi d'un ou plusieurs messages à destination des différents serveurs via le protocole PORT de bas niveau (asp_send et asp_receive). Le serveur de noms utilise le port numéro zéro, et le serveur d'entités le port numéro un.

Par exemple lors d'un port_create, il faut envoyer un message au serveur de noms pour que celui-ci retourne une capacité, puis un autre au serveur d'entités pour que celui-ci crée une nouvelle entrée dans la table des blocs de contrôle d'entités.

Le processus appelant est alors suspendu et n'est réveillé qu'au retour d'une réponse de la part du serveur. Ceci permet de donner la main à d'autres processus utilisateur, quand par exemple le serveur ne peut pas traiter immédiatement la requête.

(5) Exécution d'un ou plusieurs appels système de bas niveau et retour au niveau utilisateur. Cette étape est parfois intercalée avec l'étape (4).

6.1.3 Les serveurs de désignation

Au dessus du noyau PARX, nous avons décidé de créer sur chaque processeur, deux serveurs pour la gestion de la désignation. Un premier serveur, que nous avons appelé *serveur de noms*, se charge de construire les identificateurs d'entité et de retourner les capacités correspondantes. Le deuxième serveur, que nous avons appelé *serveur d'entités*, se charge de stocker et gérer les blocs de contrôles d'entités (ECB), et de traiter les requêtes de localisation d'entités.

Nous rappelons qu'un ECB est une structure qui contient l'identificateur de l'entité, sa localisation, et de nombreux indicateurs qui décrivent son état (privé, lecture seule, en

16 Sur le Transputer, le système de trap est réalisé de manière purement logicielle étant donné que ce processeur ne dispose pas de mécanisme matériel approprié.

mémoire, etc.). Il y a deux versions : la version courte qui est celle précédemment décrite, et la version longue (qui n'est allouée que sur le processeur sur lequel l'entité est créée) qui contient aussi des informations statistiques.

Nous avons décidé d'utiliser deux serveurs dans le but d'augmenter le degré de parallélisme de ParObj. En effet, puisque l'appel à un serveur bloque le processus qui a émis la requête, le fait de disposer d'un autre serveur permet éventuellement à un autre processus de voir sa requête satisfaite. Par exemple, lorsqu'un utilisateur exécute un `task_create()`, un message est d'abord envoyé au serveur de noms, puis, lorsque celui-ci retourne l'identificateur associé, un autre message est envoyé au serveur d'entités afin que celui-ci crée un bloc de contrôle. Pendant que ce processus est bloqué en attente d'une réponse du premier serveur, on peut très bien imaginer qu'un autre processus ait besoin d'appeler le serveur d'entités pour localiser une entité (par exemple à la suite d'un `port_lookup()`). Le fait de disposer de deux serveurs permet donc d'augmenter le temps de réponse des appels système. Ceci est résumé dans le figure 6.2.

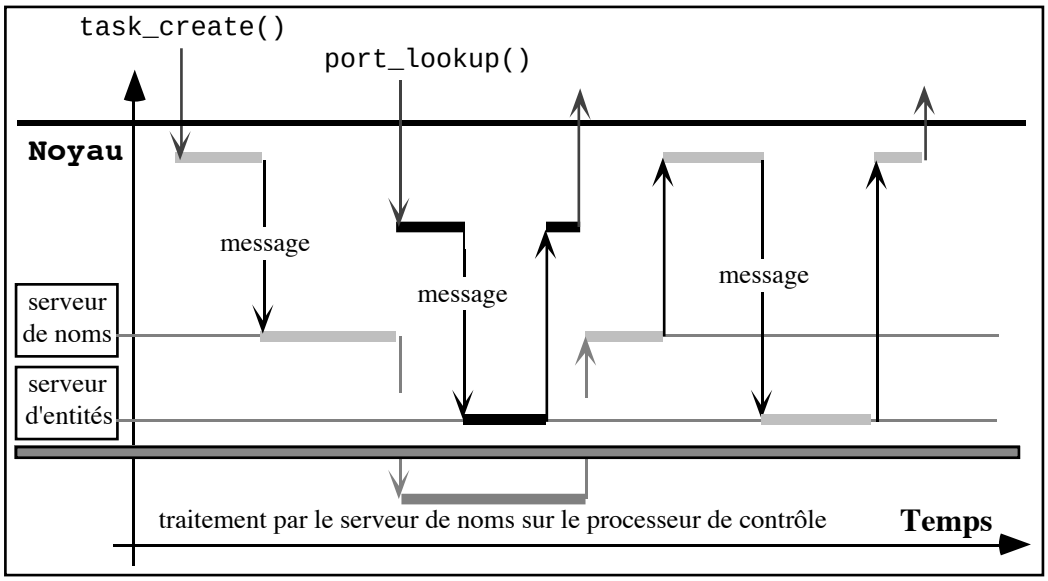


Figure 6.2. Deux serveurs pour un meilleur temps de réponse

Dans cet exemple, avec un seul serveur, le processus qui exécute le `port_lookup()` aurait dû rester bloqué en attente de la fin du traitement de l'appel `task_create()`. Avec deux serveurs, aux fonctionnalités bien différentes, nous avons augmenté le temps de réponse du système.

Lorsqu'un appel système de haut niveau s'adresse au serveur de noms, ou lorsque ce serveur s'adresse à un autre serveur de noms (parce qu'il ne peut pas satisfaire localement la requête), un message de type `t_naming_request` est envoyé. Ce message se compose de deux parties : la structure `t_naming_args` qui contient le type de la requête ainsi que des paramètres éventuels, et la structure `t_reply_loc` qui indique au serveur à qui la réponse doit être retournée (soit localement, soit sur un port d'un processeur distant).

```
typedef struct naming_request {
   t_naming_args    nArgs  ;
   t_reply_loc      nReply ;
} t_naming_request, *t_nam_req_ptr ;


typedef struct reply_loc {
   union {
      t_port_localization rReplyPort  ;/*requête distante*/
      t_address           rAddress    ;/*requête locale  */
   } u ;
   t_boolean              rLocal      ;/*VRAI si locale  */
} t_reply_loc, *t_reply_loc_ptr ;


typedef struct port_localization {
   t_processor           pProc  ;
   t_indice              pIndex ;
} port_localization t_port_localization   ;

#define rPROC             u.rReplyPort.pProc
#define rINDEx            u.rReplyPort.pIndex


/*
 *  Le numéro de port associé au serveur de noms
 */
#define N_SERVER_PORT_ID        0

typedef unsigned short      t_naming_req_type ;


/*
 *  Les requêtes possibles (Cf. fonctionnement du serveur de noms) :
 */
#define nREQ_CREATE_ENTRY     (t_naming_req_type) 0x0010
#define nREQ_GET_IDENTIFIER   (t_naming_req_type) 0x0020
#define nREQ_DELETE_ENTRY     (t_naming_req_type) 0x0030
#define nREQ_RELEASE_IDENT    (t_naming_req_type) 0x0040

/*
 *  Les deux requêtes suivantes sont utilisés par le gestionnaire d'un
 *  sous-système qui fournit un modèle de programmation dans lequel il est
 *  possible de communiquer entre processus par l'intermédiaire de variables
 *  globales,permettant ainsi de s'affranchir du paradigme de l'envoi de messages
 */
#define nREQ_CREATE_GDATA     (t_naming_req_type) 0x0050
#define nREQ_DELETE_GDATA     (t_naming_req_type) 0x0060
```

```
typedef union naming_args {
   t_naming_req_type   nType   ;

   struct {  // Pour la destruction d'entité
      t_naming_req_type nType   ;
      t_short_eid       nEntity ;/* l'Ident. de l'appelant */
   } de ;

   struct {  // Pour la création d'entité
      t_naming_req_type nType   ;
      t_entity_type     nWhat   ;/* Task, thread, port, etc. */
      t_visibility      nVis    ;/* Visibilité de l'entité   */
      t_short_eid(17)   nOwner  ;/* l'Ident. du propriétaire */
   } cr ;

} t_naming_args, *t_nam_args_ptr;
```

Le lecteur pourra remarquer que les capacités ne sont jamais envoyées aux serveurs. Ceci permet d'optimiser la taille des messages (en n'envoyant que l'EID) et réduire les temps d'attente (en évitant aux serveurs la perte de temps du cryptage/décryptage des capacités). Comme les serveurs se trouvent dans le noyau – donc inaccessibles à l'utilisateur, nous sommes sûrs que les informations concernant les entités restent confidentielles.

Le serveur de noms, que ce soit sur le processeur de contrôle ou sur les processeurs de travail, fonctionne de la manière suivante (nous appelerons serveur de noms de contrôle le serveur de noms qui s'exécute sur le processeur de contrôle) :

```
pour toujours faire
   /*
    * Lecture de la requête
    */
   asp_receive(N_SERVER_PORT_ID,
                        sizeof(LcRequest),&LcRequest) ;

   cas ( LcRequest.nArgs.nType ) dans
      nREQ_CREATE_ENTRY:
      /*
       * Création d'un identificateur d'entité. Si l'entité est une tâche
       * ou un objet actif, le serveur envoie une requête de type
       * nREQ_GET_IDENTIFIER au serveur de noms de contrôle pour connaître
       * quel identificateur il faut assigner à l'entité. Dans le cas
       * contraire (par exemple pour la création d'un thread), un numéro
       * local est assigné et aucune requête au serveur de noms de
       * contrôle n'est nécessaire.
```

17 Comme nous l'avons expliqué dans le chapitre sur la désignation, l'identificateur du domaine n'est pas utilisé lorsque l'entité est privée. Par conséquent, il n'est pas nécessaire de l'envoyer au serveur. C'est pourquoi nous utilisons la version courte de l'EID, ce qui permet de réduire le nombre d'octets émis. lorsque l'entité est publique, nous devons aussi accoller à la suite du message `t_naming_request`, le domaine manquant, ce qui se traduit par une légère augmentation du temps d'exécution des appels système, comme nous le verrons dans le chapitre **6.2 mesures de performances.**

```
        */

        nREQ_GET_IDENTIFIER:
        /*
         * Retourne un identificateur unique (au niveau de la Ptâche) pour un
         * tâche ou un objet actif. Cette requête n'est exécutée que par le
         * serveur de noms de contrôle.
         */

        nREQ_CREATE_GDATA:
        /*
         * Création d'un identificateur d'entité pour une variable globale (en
         * cours de réalisation).
         */

        nREQ_DELETE_ENTRY:
        /*
         * Indique au serveur que l'entité considérée a été détruite, les
         * informations concernant son nom peuvent donc être supprimées. Si
         * l'entité est une tâche ou un objet actif, un message de type
         * nREQ_RELEASE_IDENT est envoyé au serveur de noms du processeur de
         * contrôle pour que celui-ci mette à jour ses informations.
         */

        nREQ_RELEASE_IDENT:
        /*
         * Désalloue l'information relative au nom de l'entité spécifiée et met
         * à jour le 'compteur' d'identificateur. Cette requête n'est exécutée
         * que par le serveur de noms de contrôle.
         */

        nREQ_DELETE_GDATA:
        /*
         * Destruction de l'identificateur d'entité pour une variable globale
         * (en cours de réalisation).
         */
    fincas

    /*
     * Renvoi de la réponse
     */
    si ( LcRequest.nReply.rLocal est VRAI ) alors
        /*
         * Récupérer dans LcRequest.nReply.u.rAddress l'adresse d'une
         * structure de données t_naming_lc_reply (non détaillée ici) que
         * nous mettons à jour et qui permet de réveiller le processus
         * endormi en attente d'une réponse.
         */
    sinon
        /*
         * Nous sommes dans le cas où le serveur de contrôle renvoie une
         * une réponse à la suite d'une requête de type nREQ_GET_IDENTIFIER
         * ou nREQ_RELEASE_IDENT. Le serveur remplit une structure de type
         * t_naming_rm_reply et l'envoie à travers le réseau, sur le port
         * LcRequest.nReply.rINDEX du processeur LcRequest.nReply.rPROC.
         */
    finsi
finpour
```

De même, lorsqu'un appel système de haut niveau s'adresse au serveur d'entités, ou lorsque celui-ci s'adresse à un autre serveur d'entités (parce qu'il ne peut pas satisfaire

localement la requête), un message de type `t_service_request` est envoyé. Ce message se compose lui aussi de deux parties : le champ `t_service_args` qui contient le type de la requête ainsi que des paramètres éventuels, et le champ `t_reply_loc` qui indique au serveur à qui la réponse doit être retournée (soit localement, soit sur un port d'un processeur distant).

```
/*
 *  Le numéro de port associé au serveur d'entités
 */
#define E_SERVER_PORT_ID         1


typedef unsigned short      t_service_req_type ;


/*
 *  Les requêtes possibles (Cf. fonctionnement du serveur d'entités) :
 */
#define sREQ_CREATE_PTSK_ENTRY   (t_service_req_type) 0x0010
#define sREQ_CREATE_TASK_ENTRY   (t_service_req_type) 0x0110
#define sREQ_CREATE_THRD_ENTRY   (t_service_req_type) 0x0210
#define sREQ_CREATE_AOBJ_ENTRY   (t_service_req_type) 0x0310
#define sREQ_CREATE_BUNC_ENTRY   (t_service_req_type) 0x0410
#define sREQ_CREATE_CLUS_ENTRY   (t_service_req_type) 0x0510
#define sREQ_CREATE_PORT_ENTRY   (t_service_req_type) 0x0810
#define sREQ_CREATE_CHAN_ENTRY   (t_service_req_type) 0x0C10
#define sREQ_CREATE_OBJE_ENTRY   (t_service_req_type) 0x1010
#define sREQ_CREATE_FOBJ_ENTRY   (t_service_req_type) 0x1410
#define sREQ_CREATE_GDATA_ENTRY  (t_service_req_type) 0x1F10

#define sREQ_CREATE_EXT_REF      (t_service_req_type) 0x0020

#define sREQ_DELETE_ENTRY        (t_service_req_type) 0x0030
#define sREQ_DELETE_EXT_REF      (t_service_req_type) 0x0040

#define sREQ_LOCATE_ENTITY       (t_service_req_type) 0x0080
#define sREQ_SEND_DATA           (t_service_req_type) 0x0090
#define sREQ_RECEIVE_DATA        (t_service_req_type) 0x00A0

#define sREQ_PUBLISH_ENTITY      (t_service_req_type) 0x0100
#define sREQ_UNPUBLISH_ENTITY    (t_service_req_type) 0x0200
#define sREQ_LOOKUP_ENTITY       (t_service_req_type) 0x0300

#define sREQ_LOCATE_CHAN_ENDS    (t_service_req_type) 0x0700
#define sREQ_CONFIGURE_CHAN      (t_service_req_type) 0x0800
#define sREQ_WAKEUP_CHAN         (t_service_req_type) 0x0900
#define sREQ_SYNCHRONIZE_CHAN    (t_service_req_type) 0x0A00


/*
 *  La strcture du message :
 */
typedef struct service_request {
   t_service_args      sArgs  ;
   t_reply_loc         sReply ;
} t_service_request, *t_serv_req_ptr ;
```

```
typedef union service_args {
   t_service_req_type    sType       ;

   /*
    * Les arguments pour la création d'un bloc de contrôle d'entité
    */
   struct {
      t_service_req_type  sType       ;
      t_attribs           sAttribs    ;
      t_localization      sLoc        ;
      t_short_eid         sShort      ;
   } cr ;

   /*
    * Les arguments pour localiser une entité, pour détruire un ECB, etc.
    */
   struct {
      t_service_req_type  sType       ;
      t_localization      sLoc        ;
      t_short_eid         sShort      ;
   } mi ;

   /*
    * Arguments pour entity_publish(<capacité>,<name>,<type>) où <type>
    * permet de spécifier que l'entité est soit privée (local à la Ptâche)
    * soit publique (visible par n'importe quelle entité de la machine),
    * pour entity_lookup() et entity_unpublish().
    */
   struct {
      t_service_req_type  sType       ;
      t_visibility        sVis        ;
      t_service_name      sName       ;
      t_short_eid         sShort      ;
   } en ;

   /*
    * Arguments pour la localisation des deux extrémités d'un canal
    * bidirectionnel (task, thread, ou object actif), et pour la
    * configuration d'un canal.
    */
   struct {
      t_service_req_type  sType       ;
      t_boolean           sNeedReply  ;
      t_indice            sSendIndex  ;
      t_indice            sRecvIndex  ;
      t_short_eid         sShort1     ;
      t_short_eid         sShort2     ;
   } ch ;

} t_service_args, *t_serv_args_ptr   ;
```

Le serveur d'entités fonctionne de la manière suivante (nous appelerons serveur d'entités de contrôle le serveur d'entités qui s'exécute sur le processeur de contrôle) :

```
pour toujours faire
|  /*
|   * reception du message
|   */
|  asp_receive(E_SERVER_PORT_ID,
|                       sizeof(LcService),&LcService) ;
|
|  cas ( LcService.sArgs.sType ) dans
|  |  sREQ_CREATE_PTSK_ENTRY:
|  |  sREQ_CREATE_TASK_ENTRY:
|  |              ...
|  |  sREQ_WAKEUP_CHAN:
|  |  sREQ_SYNCHRONIZE_CHAN:
|  |     /*
|  |      * Cf. après
|  |      */
|  fincas
|
|  /*
|   * Renvoi de la réponse
|   */
|  si ( LcService.sReply.rLocal est VRAI ) alors
|     /*
|      * Récuperer dans LcService.sReply.u.rAddress l'adresse d'une
|      * structure de données t_service_lc_reply (non détaillée ici) que
|      * nous mettons à jour et qui permet de réveiller le processus
|      * endormi en attente d'une réponse.
|      */
|  sinon
|     /*
|      * Le serveur remplit une structure de type t_service_rm_reply et
|      * l'envoie à travers le réseau, sur le port
|      * LcService.sReply.rINDEX du processeur LcService.sReply.rPROC.
|      */
|  finsi
finpour
```

Détail du fonctionnement des différentes requêtes :

sREQ_CREATE_PTSK_ENTRY:
sREQ_CREATE_TASK_ENTRY:
...
sREQ_CREATE_FOBJ_ENTRY:

Création d'un bloc de contrôle version longue (ECB_l) associé à l'entité spécifiée. Si de plus l'entité est globale (visible par toutes les entités de la Ptâche), une requête de type **sREQ_CREATE_EXT_REF** est envoyée au serveur d'entités de contrôle pour que celui-ci enregistre aussi la localisation de cette entité.

sREQ_CREATE_EXT_REF:

Création d'un bloc de contrôle version courte (ECB_r) associé à l'entité spécifiée. Cette requête n'est exécutée que par le serveur d'entités de contrôle.

sREQ_DELETE_ENTRY:

Destruction de l'ECB_l associé à l'entité spécifiée. Cette requête ne peut être reçue que par le serveur d'entités locale à l'entité. Si l'entité est globale, une requête de type **sREQ_DELETE_EXT_REF** est envoyé au serveur d'entités de contrôle pour que celui-ci supprime toute réference à cette entité.

sREQ_DELETE_EXT_REF:

Destruction de l'ECB_r associé à l'entité spécifiée. Cette requête n'est exécutée que par le serveur d'entités de contrôle.

sREQ_LOCATE_ENTITY:
sREQ_SEND_DATA:
sREQ_RECEIVE_DATA:

Localisation d'une entité. Si celle-ci n'est pas trouvée localement (et si nous ne sommes pas sur le processeur de contrôle), une requête vers le processeur de contrôle est envoyée. Si l'entité existe et si elle globale (ce qui signifie que le serveur d'entités de contrôle doit posséder un ECB_r de l'entité demandée), celui-ci retourne sa localisation. Le serveur qui a fait la requête n'a plus qu'à créer un ECB_l dans lequel il stocke l'information de localisation de l'entité. De cette manière, à la prochaine requête de localisation concernant cette entité, le serveur pourra répondre immédiatement, sans faire appel cette fois-ci au serveur de contrôle.

Lorsque la requête est de type SEND ou RECEIVE des tests supplémentaires sont effectués (par exemple si l'entité est de type READ_ONLY une requête de type SEND_DATA sera refusée.

sREQ_PUBLISH_ENTITY:

Publication d'une entitée (celle-ci doit évidemment être globale). Actuellement, il est possible de publier une tâche (pour rendre l'utilisation des canaux plus aisée, Cf. requête de traitement des canaux, en attendant une interface de programmation de plus haut niveau), un port, et une variable globale (cette fonctionnalité n'est utilisée que par le gestionnaire des variables globales, et n'est pas accessible au niveau utilisateur).

Si nous ne sommes pas sur le processeur de contrôle, une requête est envoyée vers ce processeur, car c'est le serveur d'entités de contrôle qui conserve les informations concernant la correspondance <nom de l'entité> <-> <EID>. Si la requête spécifie une entité globale, la requête doit aussi être relayée vers le serveur de contrôle système (dans la version actuelle, ceci n'est pas réalisé).

sREQ_UNPUBLISH_ENTITY:

Défait l'association <nom de l'entité> <-> <EID>. Si nous ne sommes pas sur le processeur de contrôle la requête est envoyé vers le serveur d'entités de contrôle (puiqu'il est le seul à connaitre cette information).

sREQ_LOOKUP_ENTITY:

Si nous ne sommes pas sur le processeur de contrôle la requête est envoyée au serveur d'entités de contrôle. Celui-ci recherche alors l'EID correspondant au <nom> demandé. Il renvoit alors l'EID et sa localisation au serveur qui a fait la requête. Celui-ci crée alors un ECB_r décrivant l'entité, afin que les prochaines requêtes de localisation de cette entité soient résolues localement (i.e. plus d'appel au serveur d'entités de contrôle).

sREQ_LOCATE_CHAN_ENDS:

Permet de déterminer la localisation des deux extrémités d'un canal bidirectionnel. Cette requête est en fait une double requête de localisation.

sREQ_CONFIGURE_CHAN:

Permet d'établir une connexion entre les deux extrémités (tâches ou threads) d'un canal. Les deux processus qui établissent un canal entre eux, doivent chacun effectuer un `chan_create(…)`. Ceci a pour conséquence l'envoi d'une requête de configuration à chaque serveur d'entités local.

Plusieurs cas se présentent alors : si les deux extrémités du canal sont sur le même processeur, la requête peut être traitée localement et ne pose donc aucun problème. Si la tâche distante n'a pas encore envoyée de requête demandant quelle est l'autre extrémité du canal, une requête de type **sREQ_SYNCHRONIZE_CHAN** est envoyée au serveur distant. Le serveur ne reveillera alors la tâche locale que lorsqu'elle recevra une requête de type **sREQ_WAKEUP_CHAN**. Si la tâche distante a déjà envoyé une requête de synchronisation (**sREQ_SYNCHRONIZE_CHAN**), alors le serveur local met à jour son extrémité de canal, envoit une requête de type **sREQ_WAKEUP_CHAN** au serveur distant (avec comme information l'autre extrémité du canal) et reveille la tâche locale (Cf figure 6.3 pour plus de détails).

sREQ_WAKEUP_CHAN:

Permet de réveiller une tâche en attente de configuration d'un canal bidirectionnel. Si un serveur reçoit un **sREQ_WAKEUP_CHAN** cela signifie qu'il a envoyé à un serveur distant une requête de type **sREQ_SYNCHRONIZE_CHAN**.

sREQ_SYNCHRONIZE_CHAN:

Permet de demander à un serveur distant l'établissement d'une connexion entre les deux extrémités d'un canal bidirectionnel. Cette requête n'est envoyée qu lorsqu'aucune information concernant l'autre extrémité du canal n'a été reçue.

6.1.4 Un exemple pratique : la création d'un canal entre deux tâches distantes.

Pour établir un canal bidirectionnel entre deux tâches distantes, nous ne disposons pas, comme pour les ports, d'un mécanisme de publication/recherche par un nom. En effet, ce mécanisme qui est parfaitement adapté pour l'établissement de connexions N vers un, ne s'applique plus du tout pour des liaisons un vers un, typique du canal occam, puisqu'il ne permet pas de contrôler et de connaître l'émetteur de la requête.

D'autre part, nous ne voulions pas être obligés, comme c'était le cas actuellement, d'insérer des boucles d'attente dans le code utilisateur, pour s'assurer que chaque extrémité (tâche) du canal est bien initialisée. C'est pourquoi nous avons décidé de synchroniser fortement (i.e. de bloquer) ces deux tâches, et cela tant que les deux extrémités du canal ne sont pas initialisées.

La création d'un canal se fait grâce à l'appel système :

```
chan_create(&ChanCap,&ActiveEntityCap1,&ActiveEntityCap2)
```

où `ActiveEntityCap<i>` désigne soit la capacité d'une tâche uniquement, lorsque le canal est créé entre deux entités situées sur des processeurs différents, soit la capacité d'une tâche ou d'un thread, lorsque le canal est créé entre deux entités situées sur le même processeur.

Les figures 6.3 et 6.4 décrivent en détail l'établissement d'un canal entre deux entités distantes : chacune exécute, sur son processeur respectif, un `chan_create()`. Ceci amène les deux serveurs d'entités locaux à échanger un certain nombre de messages de synchronisation (sREQ_SYNCHONIZE_CHAN, et sREQ_WAKEUP_CHAN respectivement), l'envoi de ces messages dépendant du type de la requête déjà reçu.

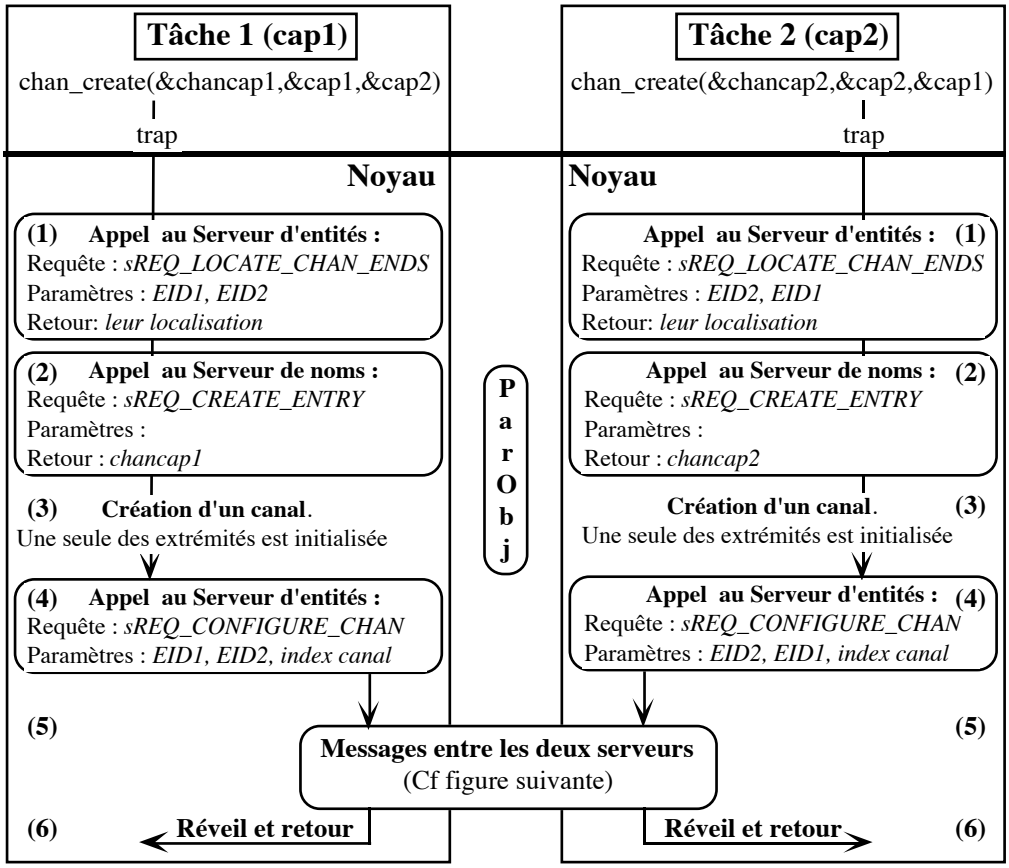


Figure 6.3. Configuration d'un canal (traitement local)

Une fois les deux extrémités du canal créées, il faut établir leurs connexions, et ne réveiller les processus que lorsque le canal est configuré. C'est le rôle de chaque serveur d'entités de s'en assurer. Nous détaillons donc maintenant, dans les deux figures suivantes, les messages échangés entre les deux serveurs, selon qu'une requête distante de synchronisation a été reçue après (deuxième cas) ou avant (premier cas) une requête locale de configuration.

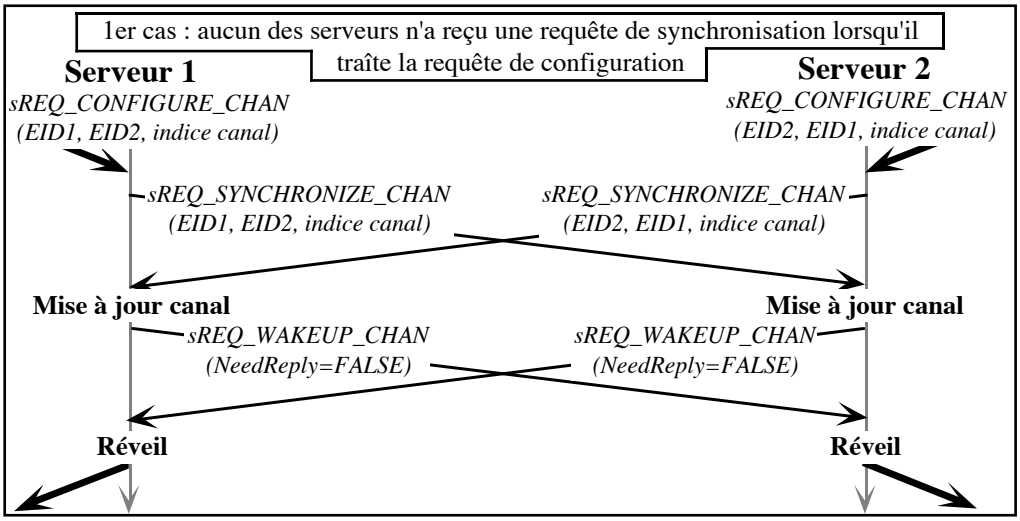


Figure 6.4a. Configuration d'un canal (messages entre serveurs — premier cas)

Comme aucun des serveurs n'a reçu de requête de synchronisation lorqu'il reçoit une requête locale de configuration, il ne connaissent ni l'un ni l'autre l'indice de l'autre extrémité du canal. C'est pourquoi chacun envoit une requête de synchronisation vers le processeur distant avec comme paramètres les EIDs des deux extrémités du canal (pour être sûr de configurer le bon canal) et l'indice local du canal. Lorsque ces requêtes arrivent, chaque serveur peut alors mettre à jour l'information concernant l'indice du canal manquant, et renvoyer un message (de type sREQ_WAKEUP_CHAN) au serveur distant pour que celui-ci puisse enfin réveiller le processus bloqué en attente de la fin de configuration du canal.

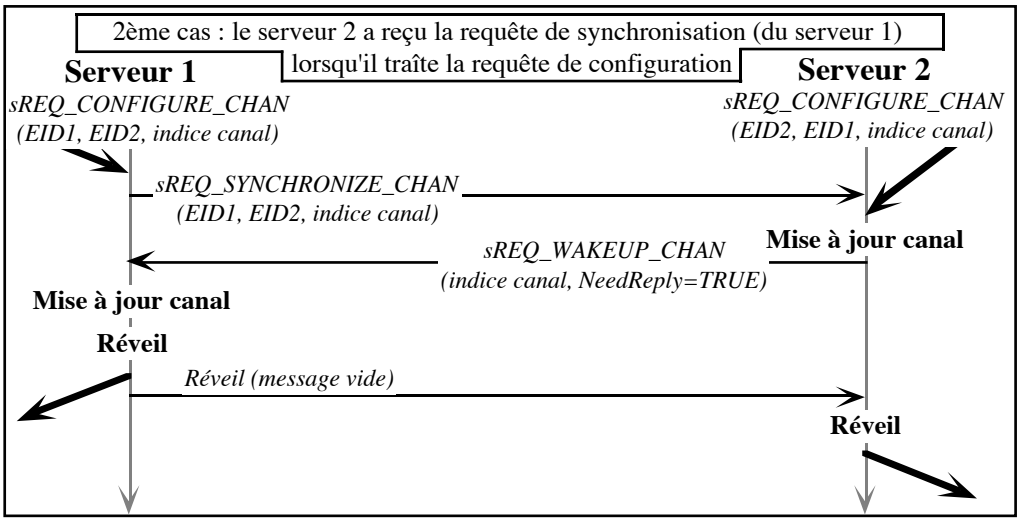


Figure 6.4b. Configuration d'un canal (messages entre serveurs — deuxième cas)

Si l'un des serveurs reçoit une requête de synchronisation avant une requête de configuration, alors lorsque celle-ci arrive, il peut aussitôt mettre à jour son canal. Dans ce cas, il renvoit une requête de type sREQ_WAKEUP_CHAN au serveur distant avec l' indice local du canal comme paramètre supplémentaire. Le serveur distant n'a plus qu'à mettre à jour son canal, à réveiller le processus bloqué, et à renvoyer un message (vide)

pour indiquer au serveur local qu'il a bien reçu l'information, et qu'il peut donc réveiller le processus endormi.

Note : la solution qui consiste à reveiller le processus local juste après l'envoi de la requête sREQ_WAKEUP_CHAN n'est pas satisfaisante, car nous ne pouvons pas garantir que le processus ainsi réveillé n'enverra pas de message sur le canal avant que l'autre extrémité du canal ne soit configurée. D'où la nécessité d'un message vide supplémentaire d'acquittement.

6.2 Mesure de performances

Dans ce paragraphe, nous avons voulu évaluer l'impact de l'implantation de la désignation sur les performances du noyau de communication PARX. Les mesures ont été effectuées sur un Supernode comportant seize transputers de travail T800 et un contrôleur T414 fonctionnant à 20 Mhz. La vitesse des liens de communication était de 10 Mbits/s.

Les mesures ont été réalisées sans imposer de charge de processeur particulière (i.e. rien de plus que de celle imposée par le noyau). En effet, des études précédentes ([Go91], [La91]) ont montré que le débit du noyau PARX était relativement peu sensible à la charge, même lorsque celle-ci était élevée (le débit diminue d'environ 12% pour une charge de 75%, et seulement de 5,7% pour une charge de 25%).

Nous avons donc plutôt chercher à déterminer :

- les temps de création des entités soit privées et statiques (l'entité n'est accessible que sur le processeur où elle a été créée), soit privées et globales (l'entité est accessible par toutes les entités de sa Ptâche), soit publiques (l'entité est accessible par n'importe quelle entité de la machine)

- le surcoût introduit par l'appel aux différents serveurs.

Nous avons réalisé les mesures de création d'entités sur le processeur zéro, en plaçant le processeur de contrôle respectivement sur le processeur zéro (cas où les temps de création sont optima) et quatre. Puis nous avons comparé les temps d'exécutions des appels système de bas et haut niveau (d'envoi de messages) entre respectivement les processeurs zéro et treize (un processeur intermédiaire), et entre le zéro et le quatorze (deux processeurs intermédiaires), tout en faisant aussi varier la position du processeur de contrôle (placé respectivement en zéro puis en quatre), Cf. figure 6.5.

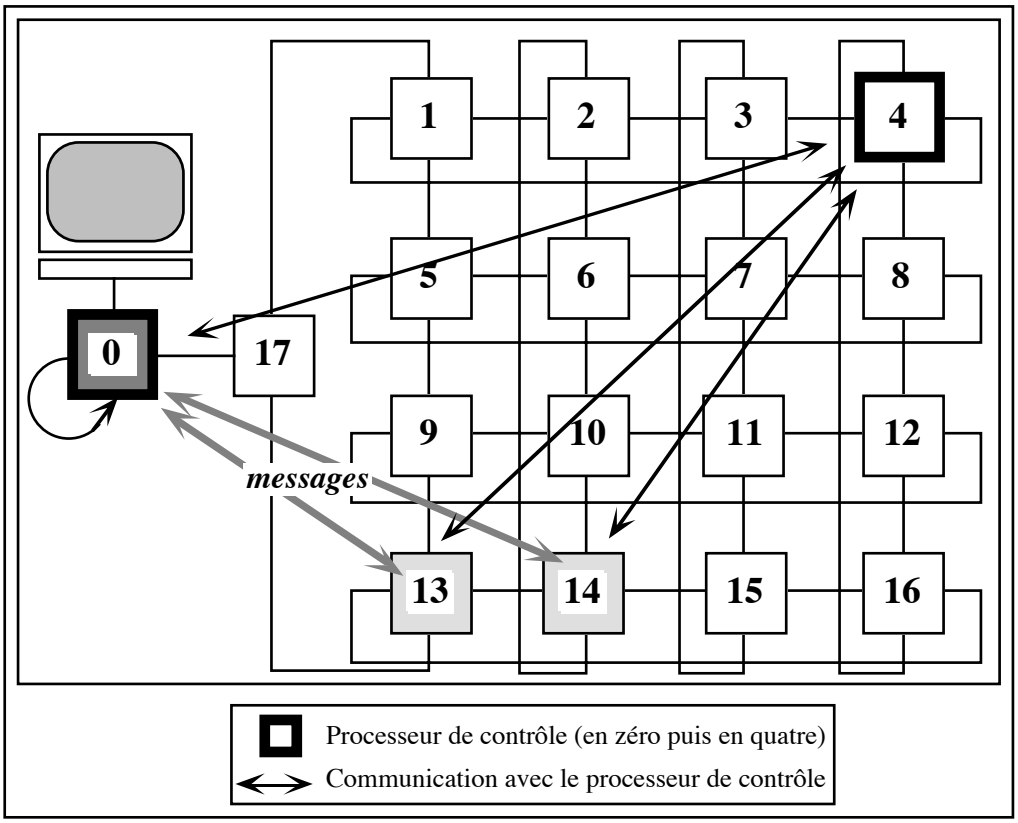


Figure 6.5. Le Supernode de travail (numéros de processeur et connexions)

Les mesures sont données à ±1 tick d'horloge, ce qui signifie une incertitude absolue de 1 µseconde en haute priorité, et de 64 µsecondes en basse priorité. Les principaux résultats sont résumés dans les différents tableaux suivants (les valeurs entre parenthèses, lorsqu'elles sont fournies, dénotent le nombre de ticks d'horloge mesurés) :

Appel système de bas niveau	***µs***
création d'un canal : *occam_chan_init*	64 (1)
création d'un port : *asp_register*	64 (1)

Appel système (haut niveau)	***Localisation du serveur***	***Entité privée et statique***	***Entité privée et globale***	***Entité publique***
task_create **(18)**	*processeur 0*	1024 (16)	1024 (16)	1344 (21)
	processeur 4	2432 (38)	3968 (62)	4608 (72)
thread_create	*processeur 0*	1344 (21)		
	processeur 4	1344 (21)		
port_create	*processeur 0*	1088 (17)	1088 (17)	1408 (22)
	processeur 4	1088 (17)	2624 (41)	3264 (51)

18 L'appel système `task_create()` est exécuté à vide i.e. aucune tâche n'est effectivement crée; seuls les appels aux différents serveurs sont réalisés.

Appel système (haut niveau)	*Localisation du serveur*	*canal entre deux tâches locales*	*canal entre deux tâches distantes*
chan_create **(19)**	*processeur 0*	2240 (35)	4032 (63)
	processeur 4	2240 (35)	4032 (63)

Le surcoût introduit lors de la création d'entité globale s'explique par le fait que le serveur local envoit une requête de type `sREQ_CREATE_EXT_REF` au serveur de contrôle pour que celui-ci puisse connaître sa localisation (à l'intérieur de la Ptâche).

Appel système (haut niveau)	*Localisation du serveur*	*Port visible au sein de la Ptâche*	*Port visible par toute la machine*
port_publish	*en 0*	704 (11)	704 (11)
	en 4	2304 (36)	2688 (42) **(20)**
port_lookup	*en 0*	576 (9)	576 (9)
	en 4	2176 (34)	2368 (37) **(5)**

De ces résultats, nous pouvons déduire que créer une entité privée et globale rajoute un surcoût d'environ 24 ticks (1536 µsecondes) par rapport à la création d'une entité statique. La création d'une entité publique rajoute encore un surcoût de 10 ticks (640 µs).

Conclusion : *notre choix de manipuler un EID réduit pour les entités privées se justifie pleinement*.

La série de mesures suivantes comparent les appels système de bas et de haut niveau de transfert de messages sur un canal (un envoi immédiatement suivi d'une réception). Les mesures de débit ont été faites pour des messages de taille respective 4 et 4096 octets, entre tâches locales et distantes.

Envoi d'un message sur un canal	**serveur**	*Message de 4 octets* *local*	*distant*	*Message de 4096 octets* *local*	*distant*
occam_chan _out/in (entre le 0 et le 13)	*en 0*	10 Ko/s (12)	5,3 Ko/s (23)	5,3 Mo/s (23)	278 Ko/s (460)
	en 4	10 Ko/s (12)	5,3 Ko/s (23)	5,3 Mo/s (23)	278 Ko/s (460)
occam_chan _out/in (entre le 0 et le 14)	*en 0*	10 Ko/s (12)	4,5 Ko/s (27)	5,3 Mo/s (23)	261 Ko/s (489)
	en 4	10 Ko/s (12)	4,5 Ko/s (27)	5,3 Mo/s (23)	261 Ko/s (489)
chan _out/in (entre le 0 et le 13)	*en 0*	3,8 Ko/s (33)	2,8 Ko/s (44)	2,1 Mo/s (61)	266 Ko/s (481)
	en 4	3,8 Ko/s (33)	2,8 Ko/s (44)	2,1 Mo/s (61)	266 Ko/s (481)
chan _out/in (entre le 0 et le 14)	*en 0*	3,8 Ko/s (33)	2,6 Ko/s (48)	2,1 Mo/s (61)	250 Ko/s (510)
	en 4	3,8 Ko/s (33)	2,6 Ko/s (48)	2,1 Mo/s (61)	250 Ko/s (510)

19 Les mesures ont été réalisées pour l'établissement d'un canal entre deux tâches situées respectivement sur le même processeur (communication locale), et situées sur le processeur zéro et quatorze (communication distante).

20 A cette valeur, il faut rajouter le coût de l'envoi du message vers le processeur de contrôle de la Ptâche système (non implanté dans cette version).

Enfin, la série de mesures suivantes compare les appels systèmes de bas et de haut niveau de tranfert de messages sur un port (un envoi immédiatement suivi d'une reception). Les mesures de débit ont été faites pour des messages de taille respective 4, 32 et 4096 octets, entre tâches distantes uniquement.

Envoi d'un message sur un port	**serveur**	***Message de 4 octets***	***Message de 32 octets***	***Message de 4096 octets***
asp_send/receive (entre le 0 et le 13)	*en 0*	7,81 Ko/s (16)	52,63 Ko/s (19)	288,0 Ko/s (434)
	en 4	7,81 Ko/s (16)	52,63 Ko/s (19)	288,0 Ko/s (434)
asp_send/receive (entre le 0 et le 14)	*en 0*	6,25 Ko/s (20)	41,67 Ko/s (24)	274,7 Ko/s (455)
	en 4	6,25 Ko/s (20)	41,67 Ko/s (24)	274,7 Ko/s (455)
port_send/receive (entre le 0 et le 13)	*en 0*	3,39 Ko/s (34)	27,03 Ko/s (37)	274,7 Ko/s (455)
	en 4	3,39 Ko/s (34)	27,03 Ko/s (37)	274,7 Ko/s (455)
port_send/receive (entre le 0 et le 14)	*en 0*	2,98 Ko/s (38)	23,81 Ko/s (42)	262,6 Ko/s (476)
	en 4	2,98 Ko/s (38)	23,81 Ko/s (42)	262,6 Ko/s (476)

De ces résultats, nous pouvons déduire que le nommage introduit un surcoût de 21 ticks environ (1344 µsecondes) avec les canaux, et de 18 ticks environ (1152 µsecondes) avec les ports, ce qui est relativement pénalisant pour les messages de petite taille (augmentation du temps de communication de 90% minimum, et jusqu'à 175%, dans le cas de communications locales). Pour les gros messages (4K), l'augmentation de temps n'est plus que de 5%, ce qui devient négligeable.

En revanche, ces résultats montrent bien que la désignation est totalement indépendante de la localisation du processeur de contrôle (puisque les serveurs sur chaque processeur jouent le rôle de cache en conservant localement les informations de localisation des entités distantes avec lesquelles le processeur communique), ce qui laisse présager des performances sensiblement équivalentes sur des machines parallèles munies d'un nombre important de processeurs (> 1000).

En conclusion, le rajout d'une désignation symbolique sur le noyau PARX, s'il fournit des fonctionnalités indispensables, n'en demeure pas moins assez pénalisant principalement lors de l'envoi de messages de petites tailles. Ceci peut se justifier par le fait que les différents serveurs utilisent le protocole port de bas niveau (`asp_receive` et `asp_send`) qui n'est pas des plus performants. Deux explications : le noyau est une version expérimentale et par conséquent son code est loin d'être optimal, enfin le transputer n'est pas assez puissant pour traiter suffisamment rapidement les messages qui arrivent sur les liens, d'autant plus que le noyau PARX utilise pleinement le CPU (actuellement, une vingtaine de processus se partagent les ressources du processeur). Il est à noter que la nouvelle génération de transputers devrait autoriser un traitement beaucoup plus efficace des transferts de messages, grâce notamment à du matériel spécialisé (C104).

Chapitre 7 : Conclusions et perspectives

Nous avons essayé, dans cette thèse, de définir les grandes lignes de ce que pourrait être un support efficace pour un système distribué à objets sur machines parallèles. Nous avons essayé le plus possible de “penser parallèle”, en essayant de réduire les goulots d’étranglement que constituent les gestions centralisées, et en exploitant au maximum les possibilités que fournit le modèle de processus à trois niveau de PARX.

L’architecture de ParObj est suffisamment générale pour supporter à la fois les paradigmes de programmation traditionnelle (à la Unix par exemple), où un programme est vu comme un simple processus, et le paradigme de programmation objet distribuée, où l’utilisateur veut pouvoir exploiter la puissance de la machine. Pour cela, il veut non seulement pouvoir disposer les différentes parties de son programme sur différents processeurs pour exploiter au maximum le parallélisme, mais aussi utiliser un formalisme à objets, qui lui permet entre autres choses, de bien structurer son application, de la rendre modulaire, concise, et de simplifier les opérations de déboggage.

Dans cette thèse, nous nous sommes penchés sur les aspects suivants : structures des entités, gestion des entités, gestion des interactions entre entités, et gestion des ressources.

De l’étude de la structure des entités, nous avons décidé que devaient être supportés aussi bien des **objets (entités) à gros grains,** tels que des fichiers, ou des grosses matrices, que des objets à **grains intermédiaires** (pour une gestion plus fine du parallélisme). Les objets à gros grains étant lourd à manipuler et diminuant fortement les accès concurrents, nous avons introduit la notion d’**objet fractionné** pour résoudre ce problème. Un objet fragmenté est un objet qui est découpé en plusieurs sous-objets indépendants (fragments de l’objet) de taille quelconque. Les fragments peuvent alors être accédés individuellement, et même en parallèle, lorsque ceux-ci sont disposés sur des processeurs différents. Les objets fragmentés permettent aussi de réduire les coûts de communication entre entités, en limitant les échanges entre processeurs et en évitant les migrations intempestives.

Ce besoin de réduire les coûts de communication, nous a amené à introduire les notions de **visibilité** d’un objet, et de **référence** sur un objet. La visibilité détermine si un objet est *privé* (sa visibilité est limitée à la Ptâche dans laquelle il a été créé), ou *public* (il peut être visible à priori par n’importe quelle entité dans la machine). La référence, quant à elle, permet de savoir si un objet est *adhérent* (il est assigner de façon permanente à un processeur), *statique* (l’objet est lié à l’entité qui l’a créé ou aux entités avec

lesquelles il communique), ou *global* (la liaison entre objets et entités peut se faire entre processeurs distants).

Ces attributs permettent l'invocation efficace d'objets (par appel de procédure lorsqu'ils sont statiques, par envoi de message dans le cas contraire), permettent au système de gérer les objets à l'intérieur de domaines bien définis, ce qui permet aux mécanismes de désignation de les retrouver plus rapidement, et de réduire le volume de données échangées entre processeurs lors de l'envoi de message.

Enfin, nous avons pensé qu'en plus du modèle original à trois processus enrichi d'objets passifs, nous avions aussi besoin de supporter des objets actifs, pour par exemple écrire des serveurs de services à forte disponiblité. L'utilisation d'objets actifs permet d'éviter l'utilisation du triplet Ptâche, tâche, thread pour leur réalisation, qui impose un coût de gestion non négligeable.

De l'étude sur la gestion des entités, nous avons déduit qu'il était indispensable de protéger les objets, afin d'éviter les accès non autorisés. PARX étant basé sur des **capacités**, nous les avons aussi adoptées pour protéger les objets de ParObj. Mais, plus important encore, nous avons décidé qu'il fallait offrir un support pour la synchronisation des objets, afin d'éliminer les conflits d'accès aux données (dûs au accès concurrents). Notre choix s'est porté sur des **scripts de synchronisation** associés aux méthodes des objets, qui permettent un contrôle des accès beaucoup plus fin que par exemple, un simple verrou en lecture/écriture.

La gestion des entités est réalisée au sein d'une Ptâche (qui est l'expression d'un programme parallèle à l'exécution). Pour chaque Ptâche, il existe une tâche particulière (qui réside dans la Ptâche de contrôle) et dont le rôle consiste à s'assurer de la bonne exécution de la Ptâche. Les différentes Ptâches ne peuvent communiquer entre elles que par l'intermédiaire de ponts de communication, ce qui permet de se protéger des accès non-autorisés.

Pour la localisation des entités, nous avons proposé un mécanisme original de désignation, qui permet de garantir les deux fonctionnalités fondamentales d'un système distribué, à savoir la transparence et l'uniformité des accès aux entités. Les identificateurs d'entités sont constitués de plusieurs champs, chaque champ définissant un identificateur local à un domaine. En fonction du contenu des différents champs, le mécanisme de désignation est capable de retrouver l'entité correspondante grâce à différents serveurs organisés de manière hiérarchique, et d'optimiser les temps d'accès lorsque les entités sont privées. Les expérimentations que nous avons réalisées ont montré que ce mécanisme est parfaitement adapté à la gestion du parallélisme massif.

Les entités sont gérées en mémoire principale de manière unique. Nous n'offrons pas à présent de notion d'objets permanents. Cependant, le système devrait pouvoir être adapté pour prendre en compte cette fonctionnalité, dans la mesure où nous avons proposé un format de fichier qui autorise le chargement d'objets à la demande.

Enfin, la gestion des processeurs est assuré grâce à la notion de **cluster**. Un cluster est un domaine de communication et de protection, constitué d'un ensemble de processeurs connectés entre eux, et dont le nombre peut varier au cours du temps. Des mécanismes pour le *placement des objets* à leur création, la *migration* et l'équilibrage *de charge* sont en cours de réalisation.

Les perspectives sont nombreuses. La première étape devra consister en l'intégration dans ParObj, de la gestion mémoire virtuelle distribuée, et du gestionnaire de fichiers parallèle, tous deux en cours de développement au sein de l'équipe SYMPA, fonctionnalités indispensables, sans lesquelles les étapes suivantes ne pourront pas être réalisées.

La seconde étape, consistera en la réalisation de la couche objet de ParObj, afin de valider les orientations choisies. En particulier, des comparaisons entre les différentes techniques développées dans cette thèse (objets fragmentés, scripts de synchronisation, etc.), et des techniques similaires utilisées dans des SDO existants (SOS, Guide, Hybrid et pSather principalement) seraient sûrement très instructives.

La dernière étape enfin, la plus ambitieuse, résidera dans la proposition de constructeurs langage, qui pourront être implantés au dessus de n'importe quel langage de programmation, et qui permettront d'augmenter le niveau d'abstraction en masquant la manière dont est réalisé le système, et d'exploiter au maximum les fonctionnalités de ParObj par la simplification de la manipulation des entités.

Cette approche "sur-langage" n'est pas nouvelle, et a déjà été adoptée dans de nombreux projets dans le monde, tels Eden [ABLN85], μC++ [BDSY92], Amber [CALL89], Linda [CG89b], etc. En effet, bien que l'approche la plus naturelle réside dans l'utilisation d'un ou plusieurs analyseurs de programmes [Eu90], [AH91] qui examinent directement le code source et extraient les informations pour la gestion du parallélisme (comme par exemple le graphe des communications entre entités [UB91], [LC91]), cette technique, quoique très prometteuse, demeure actuellement généralement incapable d'analyser n'importe quel type de programme parallèle. Aussi, il est souvent nécessaire d'utiliser soit un sous-ensemble du langage analysé, soit des constructeurs spécifiques pour aider l'analyseur ! Conclusion, l'approche "sur-langage" reste encore le moyen le plus économique et souvent le plus efficace pour programmer efficacement une application parallèle.

Ces extensions devront porter principalement sur le contrôle de type (afin de simplifier les procédures de déboggage des programmes parallèles), l'expression du parallélisme (pour pouvoir par exemple contrôler plus finement le placement sur les processeurs des différentes entités de l'application), la manipulation des objets (comme cela est déjà fait dans les langages à objets existants), et la gestion des exceptions (pour permettre le rattrapage d'erreurs, et pour augmenter la disponibilité des programmes).

IV. Annexes

A. Algorithme d'exécution d'un script de synchronisation

Ceci est l'algorithme d'exécution d'un script de synchronisation associé à une méthode d'un objet. Dans ParObj, un script de synchronisation est implanté sous forme de thread.

```
DEBUT
<prologue>:

   verrouiller l'accès à l'objet      ;
   évaluer la condition dans le EVAL  ;

   SI ( la condition est vraie ) ALORS
      SI ( EVAL est terminé par un THEN ) ALORS
         exécuter le corps du THEN ; (*)
      FINSI

      SI ( (pas de file d'attente à retard) OU
                ((une file) ET (la file est ouverte) ) ALORS
         relâcher le verrou d'accès au script  ;
         exécuter la <méthode>                 ; (**)
         aller à <épilogue>                    ;
      SINON
         // il existe une file et elle est fermée
         mettre l'<appelant> dans la file à retard  ;
         relâcher le verrou associé à l'objet       ;
         relâcher le processeur                     ;
         aller à <prologue>                         ;
      FINSI
   SINON
      // la condition est fausse
      SI ( EVAL contient une partie ELSE ) ALORS
         exécuter le corps du ELSE ; (*)
      FINSI

      SI ( pas de file d'attente à retard ) ALORS
         mettre l'<appelant> dans la file associée à l'objet ;
         relâcher le verrou associé à l'objet                ;
         relâcher le processeur                              ;
         aller à <prologue>                                  ;
      SINON
         mettre l'<appelant> dans la file à retard ;
         relâcher le verrou associé à l'objet      ;
```

```
|   |   |     relâcher le processeur                    ;
|   |   |     aller à <prologue>                        ;
|   |   | FINSI
|   | FINSI
|
| <épilogue>:
|
|   | verrouiller l'accès à l'objet   ;
|   | relâcher les éventuels verrous  ;
|   |
|   | SI ( existe une file d'attente à retard ) ALORS
|   |    POUR i DANS ( tous les éléments de la file ) FAIRE
|   |       relancer l'<appelant>[i] ;
|   |    FINPOUR
|   | FINSI
|   |
|   | POUR i DANS ( tous les éléments de la file de l'objet) FAIRE
|   |    relancer l'<appelant>[i] ;
|   | FINPOUR
| FIN
```

(*) Si le mot clé REJECT est rencontré, alors lever une <exception> et aller à <épilogue>. Si le mot clé EVAL est rencontré, relancer récursivement l'exécution du script ; en particulier, si la condition est bloquante, alors lorsque le processus sera débloqué, c'est cette condition qui sera réévaluée.

(**) Si le corps de la méthode contient des invocations à des méthodes d'un autre objet, l'<épilogue> est aussi exécuté juste avant chaque appel. Ceci permet de résoudre le cas où l'appel à une méthode distante bloque cette méthode, empêchant celle-ci d'exécuter l'<épilogue> (ce qui est une source d'interblocage potentiel). Cf [DDRR91] (paragraphe sur l'implantation de la synchronisation dans Guide) pour un exemple détaillé d'interblocage.

V. Bibliographie

[ABLN85] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden System: A Technical Review", *IEEE Transaction on Software Engineering SE-11,* No. 1 pp. 43-59, January 1985.

[ACF84] Yeshayahu Artsy, Hung-Yang Chang, Raphael Finkel, *"Charlotte : Design and Implementation of a distributed Kernel",* Computer Sciences Technical Report #554. August 1984.

[ACF87] Yeshayahu Artsy, Hung-Yang Chang, Raphael Finkel, "Interprocess Communication in Charlotte", *IEEE Transactions on Softaware Engineering*, 0740-7459/87/0100/022 © 1987 IEEE.

[Ada83] *"Reference Manual for the Ada Programming Language",* ANSI/MIL-std 1815-a. United States Department of Defense, 1983.

[Ag86] G. Agha, *"Actors : A Model of Concurrent Computation in distributed Systems",* MIT Press, Cambridge, Massachusetts, 1986 (thèse).

[AGHR89] François Armand, Michel Gien, Frédéric Herrmann and Marc Rozier, *"Revolution 89 or 'Distributing UNIX Brings it Back to its Original Virtues'",* Chorus Systèmes, rapport CS/TR-89-36.1, 1989.

[AH87] G. Agha and C. E. Hewitt, "Concurrent Programming using Actors", In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pp. 37-53, MIT Press, Cambridge, Massachusetts, 1987.

[AH91] Santosh G. Abraham, David E. Hudak, "Compile-Time Partitioning of Iterative Parallel Loops to Reduce Cache Coherency Traffic", *IEEE Transactions on Parallel and Distributed Systems, Vol. 2*, No. 3, pp. 318-328, July 1991.

[Am89] P. America, "POOL-T: a Parallel Object-Orientated Langage", In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pp. 199-220, MIT Press, Cambridge, Massachusetts.

[An90] Pascal Michel Ansquer, *"Machine virtuelle pour la compilation sur une architecture massivement parallèle",* Rapport de DEA, septembre 1990.

[Apple88] Apple, *"Inside Macintosh Volume I",* Addison-Wesley Publishing Company, Inc., December 1988.

[Ba86] Maurice J. Bach, *"The Design of the Unix Operating System",* Prentice Hall 1986.

[Ba87] R. Balter et al, *"Modèle d'exécution du système Guide",* Rapport Guide-R3, Décembre 1987, Bull-LGI-IMAG.

[BBBC88] Robert V. Baron, David Black, William Bolosky, Jonathan Chew, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr. and Michael Wayne Young, *"MACH Kernel Interface Manual",* Draft. 15 February 1988.

[BBCF92] François Barbou des Places, Philippe Bernadat, Michael Condict, Jacques Febvre, David George, James Loveluck, Éamonn McManus, Simon Patience, José Rogado, Patrick Roudaud, *"Architecture and Benefits of a Multithreaded OSF/1 Server",* A OSF Research Institute Memorandum.June 1992.

[BBK91] R. Balter, J. P. Banâtre, S. Krakowiak, *"Construction des systèmes d'exploitation répartis — Chapitre 7 : Gestion répartie d'objets",* INRIA, 1991.

[BDSY92] P. A. Buhr, Glen Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke, "μC++: Concurrency in the Object-oriented Language C++", *Softaware— Practice and Experience*, Vol. 22, No. 2, pp. 137-172, February 1992.

[BFFG89] J. Briat, M. Favre, D. Fort, N. Gonzàlez-Valenzuela, Y. Langué, T. Muntean, Ph. Waille, *"PARX: A Parallel Operating System for Transputer Based Machines",* Proc. of 10th OUG, 3-5 April, IOS, Springfield, Amsterdam 1989.

[BHLC87] A.P. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald", *IEEE Transaction on software Engineering, Vol. 13*, No. 1, pp. 65-76, January 1987.

[BKT92] H. E. Bal, M. F. Kaashoek and A. S. Tanenbaum, "Orca: a language for parallel programming of distributed systems", *IEEE Transactions on Software Engineering*, Vol. 18, No. 3, MArch 1992.

[BM89] A. C. Magalhaes de Melo Balaniuk, T. Muntean, *"Basic Mechanisms for the Supernode File System",* Supernode Project Working Paper – 0589 IMAG–LGI Laboratory, Univ. of Grenoble.

[BN89] Uwe M. Borghoff and Kristof Nast-Kolb, *"Distributed Systems: A Comprehensive Survey",* Technical Report No TUM-I8909, November 1989, Techn. Univ. Münche, Munich, Germany.

[Br85a] J.P. Briot, *"Instanciation et héritage dans les langages objets",* Thèse de 3ème cycle, Université de Paris 6, LITP 85-21, 1985.

[Br85b] J.P. Briot, "Les métaclasses dans les langages orientés objets", *Actes du 5ème CARFIA*, pp. 755-764, Grenoble, 1985.

[Br88] M. Braner, *"Trollius Manuals",* Cornel Theory Center, 1988.

[BST89] Henri E. Bal, Jennifer G. Steiner, Andrew S. Tanenbaum, "Programming Langages for Distributed Computing Systems", *ACM Computing Surveys, Vol. 21*, No. 3, September 1989.

[Bu87] Gregory D. Burns, *"Interprocess Communication In Trillium V1.0 User's Guide",* Preliminary Report, July 23, 1987.

[Ca84] L. Cardelli, "A Semantics of Multiple Inheritance", In G. Kahn, D.B. McQueen, and G. Plotkin, editors, *Semantics of Data Types*, pp. 51-67, Springer-Verlag, Lecture Notes in Computer Science, Vol. 173, 1984.

[Ca91] Harold Enrique Castro Barrera, *"Gestion de Ressources Virtuelles Partagées dans les Machines Parallèles",* Rapport de DEA d'informatique. IMAG, 20 septembre 1991.

[CALL89] Jeffry S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield, "The Amber System: Parallel Programming on a Network of Multiprocessors", *Communication of the ACM* © 1989 ACM 089791-338-3/89/0012/0147.

[CC91] R.S. Chin and S.T. Chanson, "Distributed Object-Based Programming Systems", *ACM Computing Surveys, Vol. 23*, No. 1, pp. 91-124, March 91.

[CCL89] A. Ciampolini, A. Corradi, L. Loenardi, "A Generalized Support for Parallel Object Models", 1989.

[CG86] N. Carrieri and D. Gelernter, "The S/Net's Linda kernel", *ACM Transactions on Computer Systems, Vol. 4*, No. 2, pp. 110-129, May 1986.

[CG89a] Nicholas Carriero and David Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed", *ACM Computing Surveys, Vol. 21*, No. 3, September 1989.

[CG89b] N. Carriero and D. Gelernter, "Linda in Context", *Communications of the ACM, Vol. 32*, No. 4, April 1989.

[Ch92] Pranay Chaudhuri, "Parallel Algorithms – Design and Analysis, Chapter 6 : Matrix Computations", Richard P. Brent – Editor, 1992.

[Chorus89] *"CHORUS Kernel V3.1: Specification and Interface",* Chorus Systèmes, report CS/TN-87-25.3, April 1989.

[CK91] K. Mani Chandy, Carl Kesselman, “Parallel Programming in 2001”, *IEEE Software, Vol. 8*, No. 6, pp. 11-20, November 1991.

[Da90] Dasgupta et al, “The Design and Implementation of the Clouds Distributed Operating System”, *Computing Systems, Vol. 3*, No. 1, 1990.

[DDRR91] D. Decouchant, P. Le Dot, M. Rivell, C. Roisin, X. Rousset de Pina, “A synchronization Mechanism for an Object Oriented Distributed System”, *IEEE proceeding of ICDCS Conference*, 1991.

[De90] D. Decouchant et al, *“Architecture and Implementation of GUIDE, an Object-oriented Distributed System”,* IMAG Technical Report, Janvier 1990.

[De92] Robert Despons, *“Diffusion correcte dans les machines parallèles sans mémoire commune — implémentation pour Parx”,* Rapport de DEA, 23 juin 1992.

[Di71] Dijskstra, “Hierarchical Ordering sequential process”, *Acta Informatica, Vol. 1*, No. 1, January 71.

[DN66] O.J. Dahl and K. Nygaard, “SIMULA – An ALGOL-Based Simulation Language”, *Communication of the ACM, Vol 9*, No. 9, pp. 671-678, 1966.

[Do88] L. W. Dowdy, *“On the Partionioning of Multiprocessor Systems”,* Technical Report, Dept. of Computer Science, Vanderbilt University, March 1988.

[DO91] Fred Douglis, John Ousterhout, “Transparent Process Migration: Design Alternatives and the Sprite Implementation”, *Software—Practice and Experience, Vol. 21*, No. 8, pp. 757-785, August 1991.

[Du86] P. Dugerdil, “A propos des mécanismes d'héritage dans les langages orientés objets”, *Actes du 2ème CIIA*, pp. 67-77, Marseille, 1986.

[El92] Ahmed Elleuch, “La migration de processus dans les systèmes parallèles : intérêt et faisabilité”, Rapport Interne, Décembre 1992.

[Eu90] J. Eudes, *“PDS : Un générateur de système de développement pour machines multiprocesseurs”,* Thèse. LGI-IMAG, 1990.

[FLM91] Jerome A. Feldman, Chu-Cheow Lim, Franco Mazzanti, *“pSather monitors: Design, Tutorial, Rationale and Implementation”,* TR-91-031, September 1991.

[FS78] R. Finkel and M. Solomon, *"The Roscoe Kernel",* Technical Report 337, Computer Sciences Dep., Univ. of Wisconsin-Madison, October 1978.

[FSDL83] R.A. Finkel, M.H. Solomon, D. DeWitt, and L. Landweber, *"The Charlotte Distributed Operating System",* Technical Report 502, Univ. of Wisconsin-Madison Computer Sciences, October 1983.

[Ga87] N. H. Garnett, *"Helios : an Operating System for the Transputer",* Proc. of OUG-7, IOS, Springfield, 1987.

[GB83] A. Goldberg and D.G. Bobrow, *"Smaltalk-80, the Language and its Implementation",* Addison Wesley, Reading, Masschussetts, 1983.

[Go91] Nestor Gonzalez Valenzuela, *"Parx : Noyau de système pour les ordinateurs massivement parallèles — contrôle de la communication entre processus",* Thèse. LGI-IMAG. Vendredi 13 Décembre 1991.

[GST90] Dipak Ghosal, Giuseppe Serazzi, and Satish K. Tripathi, "Processor Working Set ant its Use in Sceduling Multiprocessor Systems", *Submitted to IEEE Transaction System*, May 1990.

[GTSN90] Dipak Ghosal, Satish K. Tripathi, Guiseppe Serazzi, Sam H. Noh, Paolo Lenzi, Ashok K. Agrawalal, "Characterizing Parallel Program Behavior : An Experimental Study", *IEEE Computer System*. June 1990.

[Ha89] Habert Sabine, *"Gestion d'objets et migration dans les systèmes réparties",* Thèse Informatique. Paris 6. 1989.

[HAJ82] C.E. Hewitt, G. Attardi and P. de Jong, *"Open Systems",* AI Memo 691, AI Lab, MIT, Cambridge, Massachusetts, 1982.

[HE91] Michael T. Heath, Jennifer A. Etheridge, "Visualizing the Performance of Parallel Programs", *IEEE Software, Vol. 8*, No. 5, pp. 29-39, September 1991.

[HMA90] Sabine Habert, Laurence Mosseri, and Vadim Abrossimov, "COOL: Kernel Support for Object-Oriented Environments", *ECOOP/OOPSLA '90 Conference,* vol. 25 of*SIGPLAN Notices,* pp. 269-277, Ottawa (Canada), October 1990.

[Ho85] C.A.R Hoare, *"Communicating Sequential Processes",* Prentice Hall International series in compuer science, 1985.

[IB90] Petro Istavrinos, Lothar Borrmann, *"A Process and Memory model for a Parallel Distributed-Memory Machine",* Lecture Notes in Computer

Science N° 457, COMPAR 90-VAPP IV. H. Buskha (Ed.). Zurich, Switzerhand, September 1990.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, Andrew Black, "Fine-Grained Mobility in the Emeral System", *ACM Transactions on Computer Systems, Vol. 6*, No. 1, pp. 109-133, February 198.

[Ki88] Paul J. King, *"Implementing a POSIX Compatible Operating System on a Multi-transputer Supercomputer",* EUUG, 3-7 Oct. 1988, pp. 39-51.

[KMNR90] 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 Aplications", *JOOP* September/October 1990.

[KNP88] D. Konstantas, O.M. Nierstraszn , and M. Papathomas, "An implementation of Hybrid", In D. Tsichritzis, editor, *Active Object Environments*, pp. 61-105, Centre Universitaire d'Informatique, Université de Genève, 1988.

[Ko81] W. Kohler, "A survey of techniques for synchronization and recovery in decentralized computer systems", *ACM Computing Surveys, Vol. 14*, No. 2, June 1981.

[La91] Yves Langue Tsobgny, *"Parx : Architecture de noyau de Système d'Exploitation Parallèle",* Thèse. LGI-IMAG. Vendredi 13 Décembre 1991.

[LC91] Jingke Li, Mzrinz Chen, "Compiling Communication-Efficent Programs for Massively Parallel Machines", *IEE Transactions on Parallel and Distributed Systems, Vol. 2*, No. 3, pp. 361-376, July 1991.

[LCJS87] Barbara Liskov, dorothy Curtis, Paul Johnson, and Robert Scheifler, "Implementation of Argus", MIT Laboratory for Computer Science Cambridge, Ma. © 1987 *ACM* 089791-242-X/87/0011/01111.

[Li88] Barbara Liskov, "Distributed Programming in Argus", *Communications of the ACM*. © 1988 ACM 0001-0782/88/0300-0300.

[LM88] Yves Langué and Traian Muntean, *"Parx: A Unix-like Operating System for Transputer-based Parallel Supercomputers",* Proc. Workshop on Unix and Supercomputers, Usenix, Pittsburg, PA, September 1988.

[Lo91] C. Lozinski, "Why I need objective-C", *Jounal of Object-Oriented Programming, Vol. 4*, No. 5, September 1991.

[Me88] François Menneteau, *"Allocateur de processus sur une architecture parallèle",* Rapport de DEA Informatique, LGI-IMAG, Septembre 1988.

[MEB88] S. Majumdar, D. L. Eager, and R. Bunt, "Sceduling in Multiprogrammed Parallel Systems", *ACM SIGMETRICS*, pp. 104-113, 1988.

[MMS91] Léon Mugwaneza, Traian Muntean, Ibrahima Sakho, *"Algorithmes de Configuration des Machines Hierarchiques Supernodes",* Rapport de Recherche. Institut IMAG. RR 853-I-Mai 1991.

[MNCL89] Gérald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard, Karl Tombre, *"Les Langages à Objets. Langages de classes, langages de frames, langages d'acteurs",* InterEditions, Paris, 1989.

[Mo81] J. E. Moss, "*Nested Transactions: An approach to reliable distributed computing",* Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1981.

[MS87] D. May and R.Shepherd, *"The Transputer Implementation of Occam",* INMOS Technical Note n° 21. February 1987.

[MT88] François Menneteau, Ollivier Taramasco, *"Allocateur de processus aux processeurs",* Projet de 3ème année, ENSIMAG, Juin 1988.

[NBW87] John R. Nicol, Gordon S. Blair, Jonathan Walpole, "Operating System Design: Towards a Holistic Approach ?", *ACM Operating Systems, Vol. 21*, No. 1, pp. 11-19, January 1987.

[Ni87] O. Nierstrasz, "Active Objects in Hybrid", *OOPSLA. Special Issue of SIGPLAN Notices, Vol. 22*, No. 12, pp. 243-253, Dec 1987.

[OLS85] B. M Oki, B. H. Liskov, and R. W. Scheifler, "Reliable object storage to support atomic actions", In *ACM Proceedings of the 10th Symposium on Operating System Principles*, pp. 147-159, 1985.

[Om91] Stephen M. Omohundro, *"The Sather Language",* Draft. 1991.

[PD89] K. H. Park and L. W. Dowdy, "Dynamic Partitionning of Multiprocessor Systems", *Internaltional Journal of Parallel Programming*, 1989.

[PK90] Michael Papathomas and Dimitri Konstantas, "Integrating concurrency and Object-Oriented Programming — An Evaluation of Hybrid", In D. Tsichritzis editor,*Object Management*, pp. 229-244, centre Universitaire d'Informatique, University of Geneva, Jully 1990.

[PWCE81] G. Popek, B. Walter, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel, “LOCUS. A Network Transparent, High Reliability Distributed System”, *ACM SIGOPTS, December 1981.*

[Ra86] R.F. Rashid, “Experiences with the Accent Network Operating System”, *In Proc. Int. Seminar on Networking in Open Systems*, pages 270-295, LNCS #248, Springer, Verlag, August 1986.

[RV77] Pierre Robert, Jean-Pierre Verjus, “Toward Autonomous Descriptions of Synchronization Modules”, In B. Gilchrist, editor, *1977 IFIP Congress Proceedings* pp. 981-986, North-Holland Publishing Company.

[Sc90] Wolfgang Schröder-Prekschat, “PEACE — A Distributed Operating System for High-Performance Multicomputer Systems”, LNCS N° 433 : *Progress in Distributed Operating Systems and Distributed Systems Management*. W. Schröder-Prekschat, W. Zimneer (eds) 1990.

[Sc91] Henning Schmidt, “Making PEACE a Dynamic Alterable System”, *Lecture Notes in Computer Science*, A. Bode (Ed.), Springer-Verlag, Vol. 487, pp. 422-431, 1991.

[Sc92] Wolfgang Schröder-Preikschat, “Object Orientation in a Family of Parallel Operating Systems”, 26th Hawaii International Conference on System Science, *submitted for publication*, GMD FIRST, Berlin, Germany, 1992.

[SDPW91] Paul Sidnell, Martin Day, Andy Pepperdine, Andy Whitlow, *“Transputer common Object File Format”,* SW-0011-7, INMOS Limited confidential, 1 March, 1991.

[Se89] K. C. Sevcik, “Characterization of Parallelism in Application and Their Use in Sceduling”, *ACM SIGMETRICS Performance Evaluation Review, Vol. 17*, No. 1, pp. 171-180, May 1989.

[SGHM89] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, C. Valot, “SOS: an object-oriented operating system — assessment and perspectives”, *Computing Systems, Vol. 2*, No. 4, pp. 287-338, December 1989.

[St87] Bjarne Stroustrup, *“The C++ Programming Language”,* Addison-Wesley Series in Computer Science, Reading, Massachussetts. July, 1987.

[SuI89] Supernode I, *“Allocator and Mapper for the Supernode Program Development System”,* ESPRIT Project 1085 — SUPERNODE I. Deliverable Report, LGI-IMAG. January 1989.

[SuII91] Supernode II Workpackage 2, *"SNOS Kernel Specifications",* ESPRIT Project 2528 — SUPERNODE II. Operating Systems and Programming Environments for Parallel Computer. June 11, 1991. Final draft.

[Ta89] Andrew Tanenbaum, *"Les Systèmes d'Exploitation. Conception et mise en œuvre",* InterEditions, Paris, 1989.

[TM80] A.S Tanenbaum and S.J. Mullender, *"An Overview of the Amœba Distributed Operating System",* 1980 Mathematics Subject Classification: 68A05, 68B20. 1982 CR Categories: C.2.2, C.2.4, D.4.4, D.3.6.

[TM90] E-G. Talbi, T. Muntean, *"Placement Statique de Processus sur une Architecture Parallèle",* Rapport de Recherche. Institut IMAG. RR 833-I-Novembre 1990.

[TM92] E-G. Talbi, T. Muntean, *"Placement Dynamique de Processus sur une Architecture Parallèle",* TSI à paraître.

[TW90] T. Muntean, Ph. Waille, "L'architecture des machines Supernode", *La lettre du transputer* No. 7 (numéro spécial consacré aux machines Supernode), pp. 11-40, Septembre 90.

[TW91] A. Trew and G. Wilson, *"Past, Present, Parallel — A survey of Available Parallel Computing Systems",* Stringer Verlag, 1991.

[UB91] Theo Ungerer and Lubomir Bic, *"An Object-Oriented Interface for Parallel Programming of Loosely-coupled Multiprocessor Systems",* Universität Augsburg. Institut Für Mathzmatik. Report Nr. 223. 1991.

[Wa90] Ph. Waille, *"Introduction à l'Architecture des Machines Supernodes",* Rapport de recherche, LGI-IMAG, 26 Février, 1990.

[Wi84] N. Wirth, "History and goals of Modula-2", *Byte, Vol. 9*, No. 8, pp. 145-152, 1984.

VI. Index

T

U

V

X

[ ]

µ

f