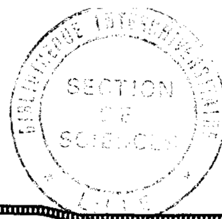


Numéro d'ordre: 320

ANNEE: 1989

USTL FLANDRES ARTOIS — LIFL — U.A. 369 du C.N.R.S.

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

50376
1989
59

50376
1989
59

THÈSE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES
DE LILLE FLANDRES ARTOIS

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

L'architecture de système réparti OMPHALE.

par

Jean-Marc Geib

Thèse soutenue le 31 Jan[illegible] xamen

Membres du Jury:

Président	: M. Dauchet,	Professeur USTLFA
Directeur	: C. Carrez,	Professeur CNAM Paris
Rapporteurs	: G. Comyn,	Professeur USTLFA
	: C. Kaiser,	Professeur CNAM Paris
Examinateurs:	M. Clerbout,	Maître de Conférences USTLFA
	E. Delattre,	Maître de Conférences USTLFA
	M. Guillemont,	Directeur de Chorus Systèmes

Université des Sciences et Techniques de Lille Flandres Artois
U.F.R. d'I.E.E.A. Bât. M3. 59655 VILLENEUVE D'ASCQ CEDEX
Tél 20 43 44 92

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PAREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CONSTANT Eugène	Electronique
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie Appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean-Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilitéss
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BREZINSKI Claude	Analyse Numérique

M. BRIDOUX Michel	Chimie-Physique
M. CELET Paul	Géologie Générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. DAUCHET Max	Informatique
M. DEBOURSE Jean-Pierre	Gestion des Entreprises
M. DHAINAUT André	Biologie Animale
M. DOUKHAN Jean-Claude	Physique du Solide
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
M. FAURE Robert	Mécanique
M. FOCT Jacques	Métallurgie
M. FRONTIER Serge	Ecologie Numérique
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. LABLACHE-COMBIER Alain	Chimie Organique
M. LACOSTE Louis	Biologie Végétale
M. LAVEINE Jean-Pierre	Paléontologie
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique Atomique et Moléculaire
M. LEROY Jean-Marie	Spectrochimie
M. LHOMME Jean	Chimie Organique Biologique
M. LOMBARD Jacques	Sociologie
M. LOUCHEUX Claude	Chimie Physique
M. LUCQUIN Michel	Chimie Physique
M. MACKE Bruno	Physique Moléculaire et Rayonnements Atmosph.
M. MIGEON Michel	E.U.D.I.L.
M. PAQUET Jacques	Géologie Générale
M. PETIT Francis	Chimie Organique
M. POUZET Pierre	Modélisation - calcul Scientifique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. SALMER Georges	Electronique
M. SCHAMPS Joel	Spectroscopie Moléculaire
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. TILLIEU Jacques	Physique Théorique
M. TOULOTTE Jean-Marc	Automatique
M. VIDAL Pierre	Automatique
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne	Composants Electroniques
M. ANDRIES Jean-Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BART André	Biologie animale
M. BASSERY Louis	Génie des Procédés et Réactions Chimiques

Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique Atomique et Moléculaire
M. BERTRAND Hugues	Sciences Economiques et Sociales
M. BERZIN Robert	Analyse
M.BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie Végétale
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie Civil
M. BOIVIN Jean-Claude	Spectroscopie
M. BOUQUELET Stéphane	Biologie Appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BRASSELET Jean-Paul	Géométrie et Topologie
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie Animale
M. CATTEAU Jean-Pierre	Chimie Organique
M. CAYATTE Jean-Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie Structurale
M. CHIVE Maurice	Composants Electroniques Optiques
M. COMYN Gérard	Informatique Théorique
M. COQUERY Jean-Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques et Sociales
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique Nucléaire et Corpusculaire
M. COUTURIER Daniel	Chimie Organique
M. CRAMPON Norbert	Tectolique Géodynamique
M. CROSNIER Yves	Electronique
M. CURGY Jean-Jacques	Biologie
Melle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	Géologie Appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean-Paul	Informatique
M. DELORME Pierre	Physioliogie Animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
M. DENEL Jacques	Informatique
M. DE PARIS Jean Claude	Analyse
M. DEPREZ Gilbert	Physique du Solide - Cristallographie
M. DERIEUX Jean-Claude	Microbiologie
Melle DESSAUX Odile	Spectroscopie de la réactivité Chimique
M. DEVRAINNE Pierre	Chimie Minérale
Mme DHAINAUT Nicole	Biologie Animale
M. DHAMELINCOURT Paul	Chimie Physique
M. DORMARD Serge	Sciences Economiques
M. DUBOIS Henri	Spectroscopie Hertzienne
M. DUBRULLE Alain	Spectroscopie Hertzienne
M. DUBUS Jean-Paul	Spectrométrie des Solides
M. DUPONT Christophe	Vie de la firme (I.A.E.)
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FAUQUAMBERGUE Renaud	Composants électroniques

M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie Sturcturale
M. GAMBLIN André	Géographie urbaine, industrielle et démog.
M. GLORIEUX Pierre	Physique moléculaire et rayonnements Atmos.
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie Physique
M. GOURIEROUX Christian	Probabilités et Statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean-Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie Organique
M. GUILBAULT Pierre	Physiologie animale
M. HENRY Jean-Pierre	Génie Mécanique
M. HERMAN Maurice	Physique spatiale
M. HOUDART René	Physique atomique
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et Statistiques
M. Jean Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme (I.A.E.)
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KREMBEL Jean	Biochimie
M. LANGRAND Claude	Probabilités et statistiques
M. LATTEUX Michel	Informatique
Mme LECLERCQ Ginette	Catalyse
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
Melle LEGRAND Solange	Algèbre
M. LEGRAND Pierre	Chimie
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne
M. LE MAROIS Henri	Vie de la firme (I.A.E.)
M. LEROY Yves	Composants électroniques
M. LESENNE Jacques	Systèmes électroniques
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique théorique
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU Jean-Marie	Optique-Physique atomique
M. MAIZIERES Christian	Automatique
M. MAURISSON Patrick	Sciences Economiques et Sociales
M. MESMACQUE Gérard	Génie Mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie Organique
M. MORTREUX André	Chimie Organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
Mme MOUYART-TASSIN Annie Françoise	Informatique
M. NICOLE Jacques	Spectrochimie
M. NOTELET Françis	Systèmes électroniques
M. PARSY Fernand	Mécanique

5

M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. STEEN Jean-Pierre	Informatique

Ce travail est aussi celui d'une équipe. Je tiens à remercier les membres de cette équipe pour leurs participations aux longues et enrichissantes discussions qui m'ont permis d'avancer dans ce travail: Christian CARREZ, Eric DELATTRE, Jean-François Méhaut et Jean-Marie PLACE.

Je remercie les membres du Jury:
Max DAUCHET pour m'avoir fait l'honneur d'accepter de présider ce jury et pour m'avoir donné dans le laboratoire qu'il dirige des conditions de travail exceptionnelles,
Christian CARREZ qui m'a pleinement accueilli dans son équipe,
mes rapporteurs Gérard COMYN et Claude KAISER qui ont rendu possible cette thèse,
Mireille CLERBOUT et ses encouragements journaliers,
Eric DELATTRE pour son aide constante,
Marc GUILLEMONT pour l'intérêt qu'il a porté à ce travail.

Je remercie ma femme Marie-Christine et ma fille Perrine , leurs présences à mes côtés est irremplaçable. Ce travail leur est dédié.

Cet exemple montre clairement
qu'il n'y a pas de trajectoire en soi
mais seulement une trajectoire par rapport
à un corps de référence déterminé.

Albert Einstein "La relativité"

Table des Matières

- **Introduction** 6
- **Chapitre 1: Le modèle Serveur** 9
    * Plan du chapitre 9
    * Objets actifs 9
    * Le Modèle Objet 10
        - *L'objet comme entité de structuration unique* 10
        - *Protocole uniforme d'accès aux objets* 12
        - *Impact de la structuration en objets* 12
    * Modèle Objet et Parallélisme 15
        - *Approche Processus et Objets* 15
        - *Approche Processus Objets* 18
        - *Le modèle Serveur* 21
    * Conclusion sur les Objets Actifs 27
    * Les systèmes répartis 27
    * Architectures réparties 28
        - *Introduction* 28
        - *Avantages et Problèmes* 29
        - *Création d'un système réparti* 31
        - *Primitives de communication* 34
        - *Transfert synchrone* 36
        - *Transfert asynchrone* 37
        - *La communication par message* 39
        - *Le nommage et la protection* 45
        - *Noms globaux* 46

- *Noms locaux* **48**

- *Capacités* **49**

* Conclusion **51**

- **Chapitre 2: Description du noyau NERF** **53**

* Le Système de Transport **55**

* Le Système d'Exécution **58**

- *Réception et Traitement* **58**

- *Emission* **59**

- *Attente sélective* **61**

- *Système d'Exécution* **62**

- *Description des Zones de Réception* **63**

- *Primitive unique d'envoi et de réception* **67**

- *Echéances* **69**

- *Primitives de gestion de la Zone d'Emission* **69**

* Nommage **71**

- *Nommages des modules* **72**

- *Nommages des services* **74**

* Protection **75**

- *Liens* **76**

- *Transfert de Liens* **77**

- *Duplication de Liens* **81**

- *Contrôle de l'utilisation des services* **82**

- *Destruction de Liens* **83**

- *Exemples d'utilisation des Liens* **85**

* Création et Destruction de modules **93**

- *Création de modules* 93

- *Primitive de création* 94

- *Destruction de modules* 95

* Conclusion sur la description de NERF 98

* Aperçu de la couche CORTEX 99

• **Chapitre 3: Deux réalisations du noyau NERF** **104**

* Architecture SPS7–SPART–ETHERNET 104

- *L'architecture matérielle SPS7* 104

- *L'architecture virtuelle SPART* 108

- *ETHERNET* 110

- *Agences spécifiques OMPHALE* 111

- *Réalisation des modules OMPHALE* 112

- *Intérêts des Modules double–tâches* 117

- *Récupération* 117

- *Conclusion* 118

* Architecture bi–processeur expérimentale 119

- *Architecture matérielle du Site 0* 119

- *Partage de la Mémoire de Modules* 121

- *Processus et Changements de contexte* 125

- *Implantation de NERF* 126

* Conclusion 130

• **Conclusion** **132**

• **Bibliographie** **135**

Introduction

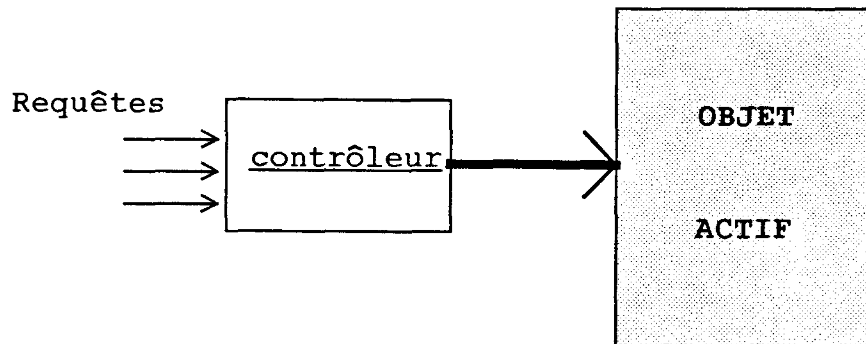
OMPHALE est un projet de création d'un **système d'exploitation réparti** du Laboratoire d'Informatique Fondamentale de Lille (Unité Associée 369 du C.N.R.S.) de l'Université des Sciences et Techniques de Lille Flandres Artois, sous la direction de Christian Carrez (professeur) et d'Eric Delattre (maître de conférences). L'équipe OMPHALE comprend aussi Jean–François Méhaut et Jean–Marie Place pour respectivement la réalisation et la simulation.

Le système OMPHALE [DEL84] [DEL85a] [DEL85b] [DEL86a] [DEL86b] [GEI86] se place dans la lignée des systèmes d'exploitation répartis conçus pour "*apparaître*" comme un système centralisé mono–processeur, alors que leurs implantations sont physiquement et logiquement réparties. Il s'agit d'obtenir les bénéfices en performances et disponibilité des architectures réparties (ensemble de machines reliées par un réseau local grande vitesse [HUT85]) dans un système d'exploitation d'ordre général comme peut l'être UNIX [RIT74] [KER84].

Cette thèse présente les principes de définition et de conception du noyau du système OMPHALE, appelé NERF, et les descriptions des plates-formes d'expérimentations réalisées.

Ce travail se fonde sur la remarque préliminaire suivante: D'une part, l'approche dite **par objets** est maintenant bien reconnue dans le domaine du Génie Logiciel comme une bonne approche structurante pour la conception des systèmes complexes, d'autre part la plupart des systèmes répartis sont structurés en termes de **processus communiquant par messages**. Ces deux approches ont des similitudes conceptuelles que nous voulons mettre en évidence.

Le but du travail présenté ici est de montrer qu'un support d'exécution d'"**objets actifs**", fondé sur un "**Modèle Serveur**" que nous définissons, peut être à la base de la conception du noyau d'un système d'exploitation réparti. Le Modèle Serveur identifie sous le nom de "**module serveur**" la **notion d'objet** au sens classique des approches par objets et celle de **processus communicant**. Ce modèle fonde toute activité complexe sur la coopération d'entités autonomes communiquant par messages de requête de service. Par rapport à la notion classique d'objet, nous munissons chaque objet actif d'un "**contrôleur** qui ordonne les arrivées asynchrones de messages . Il s'agit ici à la fois d'assurer le maintien de la cohérence de l'objet et de faire jouer l'aspect synchronisation de cet ordonnancement. Cette notion de contrôleur est directement issue de la présence des objets au niveau processus. Le contrôleur est aussi le lieu de stockage des messages en attente, stockage nécessaire pour la communication asynchrone qu'utilise notre système.



Module Serveur

Cette approche de définition de l'exécutif d'un système réparti permet de réaliser un noyau simple et uniforme. Le bénéfice attendu est dans l'utilisation du modèle Serveur pour le développement de la quasi totalité du système.

Le noyau NERF se caractérise aussi par l'utilisation de "**Capacités**" que nous appelons "**Liens**" entre modules serveurs. Ce mécanisme est bien adapté à la fois à la distribution de l'architecture ainsi qu'au contrôle de l'accès aux services des modules serveurs. Ces capacités forment donc les aspects nommage et protection de notre système.

Cette thèse se découpe en trois chapitres :

1) Définition du Modèle Serveur à partir de l'introduction du parallélisme dans le Modèle Objet et justification du Modèle Serveur en présence de répartition par l'étude des systèmes répartis existants basés sur l'utilisation de processus communiquant par messages.

2) Description détaillée du noyau NERF du système OMPHALE qui se caractérise par les aspects suivants :

- Schéma d'exécution autour d'étapes de traitement à la manière de CHORUS [BAN80] [ZIM81] [GUI82] [GUI84] [HER87].
- Communication asynchrone par messages qui sont ici des requêtes de services.
- Construction des contrôleurs sur la base d'une attente sélective de messages.
- Nommage et Protection des services par Capacités protégées dans le noyau.

- Réalisation des fonctionnalités principales d'un système d'exploitation par des "**Serveurs Systèmes**" au dessus du noyau.

3) Description des deux plates–formes d'expérimentations réalisées.

– La première consiste en une réalisation du noyau d'OMPHALE sur un réseau Ethernet de machines UNIX au dessus du noyau temps réel SPART. Outre la démonstration de faisabilité du noyau, le rôle de cette plate-forme est dans l'étude des problèmes liés à la réalisation des Serveurs Systèmes au dessus du noyau. Cela constitue les travaux actuels de Jean–François Méhaut pour sa thèse de doctorat en Informatique.

– La deuxième consiste en la construction d'une architecture spécialisée pour le support matériel du noyau d'OMPHALE. La motivation est ici la suivante : Une conséquence du modèle Serveur est qu'un support d'exécution de ce modèle doit prendre en charge de très nombreux processus et accepter de très fréquents changements de contexte. Une réponse possible, pour garder des performances acceptables, est dans la recherche de dispositifs matériels améliorant le temps de changement de contexte et plus généralement dans des architectures matériellement adaptées aux exécutifs qu'elles acceptent. Le "**Site 0**" est une architecture expérimentale d'un site, spécifiquement réalisée pour le support du noyau OMPHALE. Jean-Marie Place travaille actuellement sur des mesures de performances de cette machine. Des mesures de performances sont actuellement réalisées par Jean–Marie Place.

Chapitre 1.
Le Modèle Serveur

1.1 Plan du chapitre

Le travail de cette thèse est de mettre en place le parallélisme et la distribution dans le modèle objet [COX86] [STE86] en vue de la réalisation d'un système d'exploitation réparti. L'idée principale est ici de mettre en évidence les similitudes conceptuelles entre la notion d'objets et celle de processus communiquant par messages.

- Dans une première partie, en partant de la notion classique d'objet, nous montrons les possibilités d'introduction du parallélisme dans l'univers objet. Nous définissons à partir de là ce que nous appelons le Modèle Serveur.
- Dans une deuxième partie, à partir de l'étude des systèmes répartis existants basés sur l'utilisation de processus communiquant par messages, nous justifions le Modèle Serveur en présence de la répartition. Pour les aspects nommage et protection dans les systèmes répartis, nous utilisons un mécanisme de "*Capacités*" [LEV84] entre Serveurs.

1.2 Objets actifs

Cette partie introduit la notion d'**OBJETS ACTIFS**. Le domaine qui nous intéresse ici est celui des méthodologies de conception des systèmes (au sens large) complexes comme une collection de **MODULES** s'exécutant en parallèle et interagissant par envois de message. L'existence de ce domaine est motivé par:

1) Dans la recherche de **méthodes de conception** nouvelles pour satisfaire aux énormes demandes de systèmes de plus en plus complexes, aux fonctionnalités de plus en plus évoluées. Ces systèmes sont de plus en plus difficiles à concevoir, réaliser et maintenir. Les approches nouvelles (modulaires, par objets) sont une réponse à ces problèmes.

2) Dans la recherche de l'**exploitation du parallélisme et de la distribution** existant dans les architectures matérielles réparties. Les architectures matérielles réparties sont maintenant au centre de la recherche en création de systèmes d'exploitation d'usage courant, comme l'est Unix; et cela est en partie dû aux

progrès des techniques de communication et à l'émergence des réseaux locaux.

A l'heure actuelle, la recherche dans la création des systèmes d'exploitation répartis (c'est à dire prévus pour les architectures matérielles réparties) se place essentiellement au niveau de l'exploitation des possibilités des architectures matérielles sous-jacentes (parallélisme, disponibilité...). Le but recherché est avant tout dans l'obtention de performances permettant a posteriori de valider de telles architectures. Le système V–KERNEL est, à ce point de vue, significatif: Tous les mécanismes de base de ce système ont été définis en terme de gain d'efficacité, comme par exemple le transfert de message par un registre du processeur [CHE84b]. De même, AMOEBA [REN88] s'annonce comme le système réparti le plus rapide du monde et met en avant la recherche de performances.

Néanmoins, la recherche dans ce domaine, ne doit pas oublier la recherche sur les méthodologies nouvelles de conception, et cela, à la fois pour ses besoins propres (développement du système), mais aussi pour l'interface que ces systèmes proposeront aux développeurs d'applications. Dans ce domaine, l'approche par objets est souvent reprise, on la retrouve dans les systèmes Emerald [BLA86] [BLA87], GUIDE [BAL86], Eden [LAZ81] [BLA85] [ALM85], SOS [SHA87], Argus [LIS87] [LIS88], MACH [ACC86] [JON86] . Cette approche est cependant fort différente d'un système à l'autre, comme on le verra plus loin.

Dans ce sens, le but global d'OMPHALE et de montrer que l'approche par **OBJETS ACTIFS** et sa réalisation dans le **MODELE SERVEUR** est une approche simple et uniforme comme base d'un exécutif réparti orienté objet [GEI86].

Nous présentons d'abord le Modèle Objet qui est à la base de notre approche.

1.3 Le Modèle Objet

Les principes de base du modèle objet sont i) l'objet comme élément de base de structuration et ii) le protocole uniforme d'accès aux objets grâce à l'envoi de message.

1.3.1 L'objet comme entité de structuration unique

L'un des buts essentiels de toute conception est d'utiliser au mieux les ressources matérielles et logicielles de l'architecture sous–jacente. Le modèle Objet conceptualisé par A.K.Jones [JON79] fut un concept déterminant pour la construction structurée et fiable des systèmes complexes.

L'idée principale est la suivante : Tout composant d'un système se doit de posséder une certaine "**intégrité**". C'est à dire qu'il ne doit se comporter ou être

manipulé que dans le respect de certaines règles invariantes assurant que le composant se trouve constamment dans un état cohérent vis-à-vis de lui–même et du reste du système.

Pour mettre en oeuvre cette idée, le modèle Objet préconise :

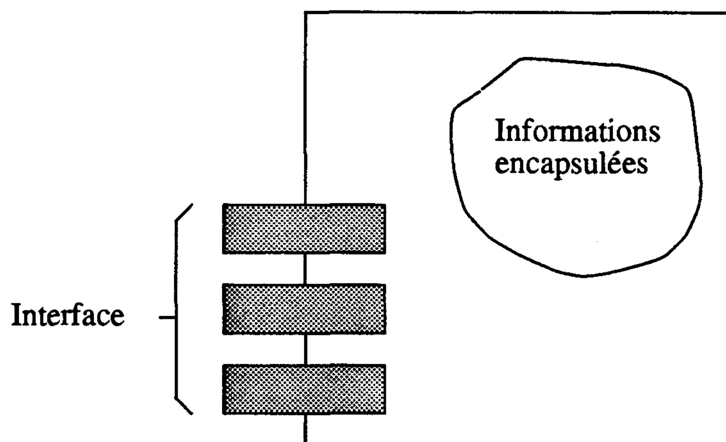
i) La structuration du système en terme de composants ayant chacun une **consistance logique** (les objets).

ii) L'application du principe d'**encapsulation.**

Un objet est un composant (vu comme un ensemble d'informations) muni d'un ensemble d'opérations qui doivent être utilisées pour le manipuler. Ces opérations, issues de la conception de l'objet, sont telles que l'état (et l'évolution de l'état) des informations va respecter les propriétés abstraites caractérisant l'objet.

La différence est immédiatement à faire ici avec la notion de bibliothèque de sous–programmes. Alors qu'une bibliothèque de sous–programmes est une sorte de fourre–tout regroupant des procédures déjà écrites qui ne sont pas obligatoirement logiquement liées, un objet a, quant à lui, un état (une représentation) qui le caractérise et les opérations qu'il offre sont fortement liées. Elles sont responsables du maintien de la cohérence de cet état.

Un objet ne doit être manipulé que par l'intermédiaire des opérations fournies explicitement. Cela constitue son "**interface**". Elle seule assure le respect de la consistance logique de l'objet. L'interface précise le "**mode d'emploi**" de l'objet qui répond à sa spécification.



Un Objet

Seules les informations et opérations utiles à ce mode d'emploi sont visibles par

l'interface, les détails de représentation, ainsi que les opérations à usage interne, sont cachées à l'intérieur de l'objet (on dit encapsulées). La représentation et les opérations d'un objet peuvent faire appel aux interfaces d'autres objets. Cette utilisation (si elle est interne) peut elle-même être encapsulée et l'on a là un moyen puissant de création de sous-systèmes : Un sous-système complet aussi complexe soit-il peut être caché derrière une interface d'accès. Cela favorise la stratification des systèmes par la décomposition en couches de plus en plus conceptuelles.

1.3.2 Protocole uniforme d'accès aux objets

Dans un système décomposé en termes d'objets, la mise en place d'une application se fait en organisant l'interaction entre les objets. Depuis surtout l'initiative de SMALLTALK [GOL83], celle-ci se définit généralement en terme d'**envois de message**. L'opération primitive d'envoi de message consiste à préciser l'objet "*destinataire*", l'"*opération*" invoquée chez le destinataire et le "*contenu*" du message qui regroupe les arguments du message. La conséquence d'un envoi de message est l'exécution de l'opération demandée dans le contexte de l'objet destinataire et la fabrication d'un résultat valeur de l'envoi de message. L'envoi de message est donc ici un appel de procédure pour lequel le nom de procédure est "*préfixée*" par un nom d'objet.

Dans SMALLTALK, où tout est objet, l'envoi de message est la seule instruction. Tout est ensuite réalisé par des opérations définies dans des objets. L'intérêt de cette approche est simple et uniforme. Elle se trouve ici bien adaptée avec l'environnement de programmation du langage, basé sur un "*browser*" [GOL84] des objets et des classes qui les définissent.

L'idée forte de l'envoi de message est la conséquence du choix de l'objet comme entité de structuration unique: Dans un univers d'objets, les objets se doivent d'avoir un comportement de base commun. Puisque tout est réalisé par interaction entre objets, il est judicieux de définir le comportement commun des objets vis à vis de leurs interactions, et, comme il est fait ici, de définir un mode d'accès unique (l'envoi de message) et uniforme (applicable à tous les objets) des entités du système. Cette manière de faire permet de séparer les problèmes de définition des objets des problèmes de l'organisation d'une application en objets.

1.3.3 Impact de la structuration en objets

Le concept d'objet a un impact considérable sur le **Génie Logiciel.** Il apporte des améliorations à la fois comme méthode structurante lors de la définition d'un système [BOO83], comme méthode efficace lors de la conception (par la réutilisabilité) [MEY88], et aussi pour l'aspect qu'il donne au système lui-même [GOL84].

i) En cours de **Définition**, il s'agit de raisonner en terme d'abstraction et de recherche de composants abstraits. Cela demande toujours une réflexion préliminaire qui doit identifier les entités intervenantes, leurs relations et la façon de les manipuler. Cette réflexion facilite fortement l'organisation et le suivi du travail de conception.
La mise au point et la validation sont elles–aussi facilitées car un composant est une entité consistante qui peut et doit être testée avant d'être intégrée dans l'application.

ii) La plupart des systèmes par objets, fournissent une collection importante de types d'objets prédéfinis. La conception ne se fait donc plus à partir du simple langage, mais en utilisant ces types d'objets prédéfinis, c'est la **réutilisabilité**. En génie logiciel, pour que la réutilisabilité soit effective, il faut avoir les moyens d'écrire du logiciel réutilisable, c'est à dire indépendamment des applications qui l'utiliseront. L'approche par objets est sûrement une des méthodes les plus adaptées pour la réutilisabilité.

iii) Les couches élevées d'un système sont souvent disponibles à l'utilisateur ou à ses programmes d'application. Lorsque le modèle objet est utilisé, l'utilisateur a donc à sa disposition un certain nombre d'interfaces qui représentent les ressources mises à sa disposition. L'utilisateur utilise donc lui–aussi, dans son travail courant, le système comme un ensemble d'abstractions. On peut citer ici le travail autour de PCTE (Portable Common Tool Environment)[ESP86] qui offre autour d'UNIX un système évolué de gestion d'objets et de leurs relations (à partir du modèle Entité–Relation issu des travaux sur les bases de données). PCTE présente aux utilisateurs une machine virtuelle sur laquelle toute interaction se réalise par dialogue avec des objets.

Les langages de programmation modernes ont bien sûr intégré la notion d'objet. Ils ont introduit les nécessaires mécanismes de composition des abstractions que sont les objets, en cela ils offrent donc des méthodologies de conception par réutilisation du logiciel.

On peut citer :

1) Le langage ADA [DOD84] se base sur la modularité (Package), l'approche par objets utilise ici ce langage par les possibilités de création de **modules génériques** (c–a–d paramétrables) [BOO83]. La création d'un composant consiste alors à paramétrer comme on le désire un "**moule**" préexistant.

2) Le système SMALLTALK [GOL83] est construit sur le concept d'objet (ici "*tout est objet*"). La programmation consiste à définir de nouveaux objets et à organiser leurs interactions. Le mécanisme de conception est ici l'**héritage** : la création d'un nouveau type d'objet se fait par spécialisation de types existants. Un progrès important de SMALLTALK a été de fournir un environnement de programmation complet écrit dans le langage lui–même, dans lequel on trouve des outils nouveaux liés au domaine des objets tel des "*inspecteurs*" ou des

"fouineurs" d'objets permettant d'examiner, d'essayer et de modifier, dans un mode très interactif, les objets existants.

3) Le langage d'**acteurs** de Hewitt [HEW77] [AGH87] est aussi un langage construit sur le concept d'objet. Les objets sont ici des entités autonomes pouvant potentiellement s'exécuter en parallèle. La définition de nouveaux objets se fait ici grâce à un mécanisme de **délégation**: Un nouvel objet peut être défini à partir d'un autre en ajoutant de nouvelles fonctionnalités. Les fonctionnalités de l'objet père peuvent être obtenues par l'objet fils en redirigeant les messages de requête de ces fonctionnalités vers l'objet père.

Tous ces mécanismes veulent offrir, dans un monde d'objets, des méthodes de conception adaptées. Ils permettent tous la définition de nouveaux objets à partir d'objets préexistants. A titre d'exemple, on peut montrer en ADA la définition d'une nouvelle abstraction de liste (on ajoute des possibilités d'entrées-sorties) à partir d'une abstraction préexistante de liste :

```
WITH liste;        -- on va heriter du package liste
                   -- pour creer un package liste avec possibilites
                   -- d'entree sortie.
GENERIC
                   -- on recupere le parametre generique de liste
  TYPE elem is private
                   -- on ajoute les procedures generiques d'entree
                   -- sortie des elements de liste
  WITH PROCEDURE get ( e : out elem ) is <>;
  WITH PROCEDURE put ( e : in elem ) is <>;


-- specification du nouveau package
PACKAGE liste_io;


                   -- on herite des fonctionnalites de liste par
                   -- instanciation du package generique liste
  PACKAGE liste2 is NEW liste(elem);


                   -- on ajoute les nouvelles fonctionnalites
                   -- d'entree sortie de liste
  PROCEDURE get_liste (l : out liste_type);
  PROCEDURE put_liste (l : in liste_type);


END liste_io;
```

ADA: Création d'un nouveau package (héritage par instanciation)

1.4 Modèle Objet et Parallélisme

Les recherches sur les architectures (matérielles et logicielles) nouvelles mettent toutes en avant des besoins d'exploitation du parallélisme. La conjonction du modèle objet et du parallélisme se place dans ces recherches pour valider les apports du modèle objet en présence de parallélisme.

Il y a plusieurs façons d'introduire le parallélisme dans le monde objet. La première, qui différencie les notions de processus et d'objets, est "*classique*" dans le sens où elle conserve la dichotomie classique des systèmes entre processus et données. La deuxième, plus novatrice, fait en sorte qu'à chaque objet soient associés un ou plusieurs processus qui réalisent les fonctionnalités de l'objet. Les objets communiquent alors au niveau inter–processus. Cette deuxième approche fait l'objet des recherche, à la fois, dans le domaine des Langages Orientés Objets parallèles [YON87], et dans la recherche de schémas d'exécution sur des architectures matérielles réparties. Les conséquences de la répartition sont analysées plus loin dans ce chapitre.

OMPHALE utilise un modèle qui unifie les notions d'objet et de processus. Le système est logiquement découpé en processus dits serveurs, un processus serveur étant une réalisation autonome d'un objet.

Nous détaillons d'abord la première approche que nous appelons **Processus et Objets**. Cette approche est celle utilisée par les systèmes Emerald, GUIDE, SOS. Et ensuite la deuxième, que nous appelons **Processus Objets** utilisée par Argus, Eden , MACH et OMPHALE.

1.4.1 Approche Processus et Objets

Cette approche se fonde sur l'aspect structuration de données de l'approche orientée objet : l'ensemble des données d'un système est découpée en objets, chaque objet regroupant une petit ensemble de données logiquement liées. Un objet est ici une entité passive (à la limite purement syntaxique).

Un processus (entité active) utilise un objet en faisant un appel, de type procédural, dans l'interface de l'objet et en exécutant le code de l'opération. Ce qu'il est intéressant de noter est que, pendant l'exécution de l'opération, la visibilité du processus est limitée au domaine de l'objet dans lequel il se trouve. Il y a donc ici en général changement de contexte d'exécution à chaque "*saut*" d'objet.

En SMALLTALK, système mono–processus, c'est le seul processus utilisateur

qui exécute l'application en "*sautant*" d'objets en objets au gré des envois de message (terme employé en SMALLTALK pour désigner l'appel d'une opération d'un objet).

A partir de là, il est possible d'exploiter ces objets (données) par plusieurs processus pouvant s'exécuter en parallèle et cela avec la même puissance que les langages classique utilisant le parallélisme. Mais on ne peut ici parler de fusion entre les idées de parallélisme et d'objets. En effet , dans ce cas, l'approche par objets n'apporte aucune solution générale aux problèmes classiques de l'exploitation du parallélisme, tels la communication et la synchronisation. Ces problèmes doivent ici être résolus par l'introduction dans le modèle de mécanismes particuliers réglant la synchronisation (et ici en conséquence la communication) des processus. On trouve souvent utilisés des mécanismes de sémaphores ou de moniteurs [HOA74].

Le problème à affronter ici est effectivement celui des objets partagés. Un objet est dit partagé s'il est accessible à plusieurs processus. Ces objets servent le plus souvent au partage d'informations entre processus et jouent un rôle de données partagées. Il est alors nécessaire de synchroniser les processus partageant un objet pour éviter les conflits d'accès aux informations qu'il renferme.

Dans le système SMALLTALK (système monoprocesseur), il est possible de créer dynamiquement des processus par une opération explicite de lancement. Les différents processus se partagent l'espace unique des objets SMALLTALK, ces processus doivent se synchroniser par un mécanisme de très bas niveau à base de sémaphores. L'envoi de message permet à un processus d'appeler une opération d'un objet, mais ne permet pas , ce qui semble paradoxal, d'envoyer un message à un autre processus. La communication doit se faire par l'intermédiaire d'objets partagés dans lesquels la synchronisation (par sémaphores) réglera l'accès à un tampon de communication. Ainsi donc SMALLTALK ne considère pas les processus comme des objets mais simplement gère la présence de plusieurs flots d'exécution (dichotomie entités actives et passives).

Le concept de moniteurs [HAN78] [HAN81] peut être utilisé pour régler le partage d'objet. En effet un moniteur peut être syntaxiquement vu comme un objet au sens précédent : Un moniteur permet l'encapsulation et définit une interface procédurale. Le mécanisme est alors classique : il assure qu'il n'y a jamais deux processus actifs simultanément à l'intérieur de l'objet; les processus peuvent être suspendus à l'entrée du moniteur (si le moniteur n'est pas libre) ou volontairement à l'intérieur du moniteur sur "*conditions*".

```
OBJET MONITEUR controleur_tampon;
  DEFINITION
    tampon : CHAINE := '';
  OPERATIONS
    deposer(CHAINE);
    retirer(CHAINE);
  REALISATION
    plein, vide : CONDITION;
    PROCEDURE deposer(ch : CHAINE);
      DEBUT
        SI longueur(tampon) > 0 ALORS vide.ATTENDRE;
        tampon := ch;
        plein.SIGNAL;
      FIN;
    PROCEDURE retirer(ch : CHAINE);
      DEBUT
        SI longueur(tampon) = 0 ALORS plein.ATTENDRE;
        ch := tampon;
        tampon := '';
        vide.SIGNAL;
      FIN
FIN MONITEUR;
```

Moniteur pour le contrôle d'un tampon

Le langage et système Emerald est un exemple de d'exploitation du parallélisme par un modèle Processus et Objets. Toute les informations sont encapsulées dans des objets , et les objets ne peuvent être manipulés que par invocation des opérations qui les caractérisent. L'exécution se fonde sur la notion de processus, et donc, en cas d'objets partagés, plusieurs processus peuvent s'exécuter simultanément à l'intérieur d'un objet. Pour synchroniser ces processus, un objet peut préciser une "*section moniteur*" permettant d'obtenir l'accès exclusif aux variables protégées et la synchronisation par conditions.

Dans ce système, les processus sont créés à l'initialisation d'objets particuliers, ceux possédant une "*section processus*". Ces objets sont appelés Objets Actifs par Emerald. Il ne s'agit néanmoins pas ici de ce que nous appelons l'approche Processus Objet, car un tel processus issu d'un objet va évoluer d'objet en objet au gré des déclenchements d'opérations. Les objets actifs de Emerald ne sont que les initiateurs des processus du systèmes et non pas des processus objets.

Les systèmes SOS et GUIDE se fondent eux aussi sur des objets passifs et des processus. Dans ces deux systèmes, un processus s'exécute dans un contexte d'exécution (appelé "*domaine*" dans GUIDE) qui référence des objets. Lors de l'appel d'une opération d'un objet en dehors du contexte, soit celui–ci est ajouté au

contexte (GUIDE) soit reçoit un "*mandataire*" de l'objet(c'est–à–dire un objet particulier représentant de l'objet). Dans le système SOS, l'appel d'une opération d'un mandataire entraîne un changement de contexte du processus. Dans ces systèmes, un objet peut appartenir à plusieurs contextes. La synchronisation nécessaire est alors réalisée par des mécanismes classiques de sémaphores ou moniteurs.

On reparlera de ces deux derniers systèmes dans le cadre de la répartition.

1.4.2 Approche Processus Objets

Dans cette approche les objets deviennent des entités actives (assimilables à un processus ou un groupe de processus) qui vont communiquer de manière uniforme par un mécanisme d'envoi de message au niveau inter–processus. On voit tout de suite l'intérêt de cette approche par rapport à la précédente:

1) L' objet actif est l'unique entité de structuration du système, la dichotomie actif/passif n'apparaît plus à la base du système.

2) L'espace inter–processus (où transite les messages) sert à l'interaction entre objets actifs. L'envoi de message sert à la communication et peut donc être la base de la synchronisation des objets actifs sans l'introduction de mécanismes supplémentaires.

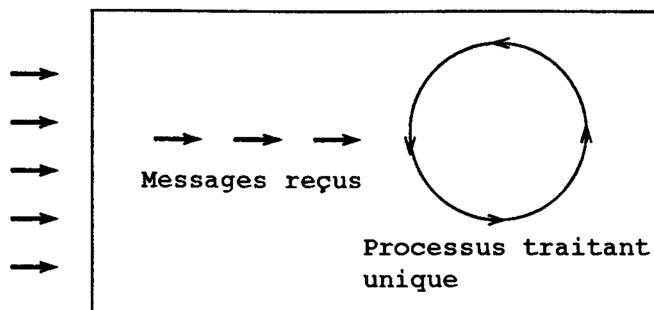
On retrouve donc ici les avantages de l'approche classique par objets : Entité unique de structuration qui encapsule des données et offre une interface officielle d'appel, protocole uniforme d'accès aux objets par envoi de message. Cependant le schéma d'exécution est ici changé: L'exécution d'une application est réalisée par les exécutions concurrentes des différents objets.

Tous ces objets sont ici plongés dans un espace inter–processus où transitent les messages, et donc un objet va devoir répondre aux différents messages qui lui arrivent et cela de façon asynchrone (ce terme est ici employé pour indiquer qu'un message peut arriver à un objet alors que celui–ci n'est pas en attente de message). Le schéma d'exécution doit s'adapter à cet état des choses. Techniquement deux solutions apparaissent: Les processus objets monoprogrammés et les processus objets multiprogrammés.

1.4.2.1 Processus Objets monoprogrammés

L'objet actif est supporté par un **processus séquentiel unique** qui va répondre aux messages reçus. Ceux–ci sont stockés à l'entrée de l'objet et sont donc traités séquentiellement. L'ordre de traitement des messages reçus peut être celui d'arrivée, ou sur la base de priorités, ou encore sur la base d'un algorithme interne à l'objet. On verra qu'OMPHALE se rattache à cette solution. Nous détaillerons cette solution plus loin sous le nom de Modèle Serveur. Dans cette approche nous

avons l'équation: un objet = un processus, nous l'appelons solution **Processus Objets monoprogrammés.**



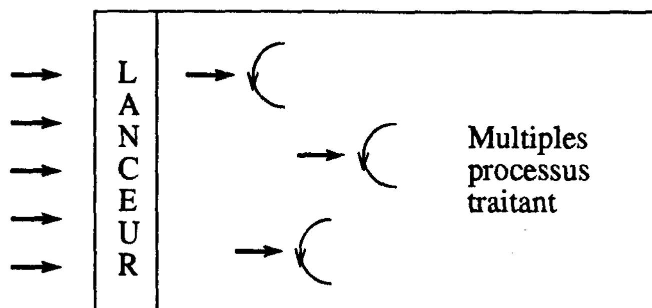
Objet Actif Monoprogrammé

1.4.2.2 Processus Objets multiprogrammés

L'objet actif est supporté par un ensemble de processus. Chaque processus est créé dynamiquement pour répondre à un message reçu. Ces processus partagent le même contexte d'exécution (que l'on peut assimiler en première approximation à l'espace mémoire correspondant à l'objet). Ils sont alors appelés "*processus légers*" ou "*activités*" ("*thread*"). La notion de processus légers vient du découpage de la notion de processus en i) ce qui représente son activité (vecteur d'état et pile d'exécution) et en ii) son contexte d'exécution (pages mémoires accessibles). Le changement de contexte est un mécanisme lourd, le changement d'activité est une opération bien plus simple. Ici il est intéressant d'avoir des activités se partageant le même contexte d'exécution. La gestion de multiples activités à l'intérieur d'un objet actif est alors une technique abordable vis-à-vis des performances attendues. Ce mécanisme est la solution actuelle au support d'objets actifs multi-processus: l'utilisation de processus classiques (comme le sont ceux d'Unix) entraînerait une charge trop importante en gestion de ces contextes.

Avec ce dernier mécanisme, il existe dans chaque objet un processus particulier (que nous appellerons "**lanceur**") qui se trouve "*à l'écoute*" des messages qui arrivent et crée en conséquence les processus qui vont traiter ces messages. Chaque processus lanceur traite séquentiellement les messages qu'il reçoit en créant un processus léger correspondant. La synchronisation apparaît ici donc entre les processus légers d'un objet: elle peut se fonder sur une communication par messages ou par des techniques classiques de sémaphores ou moniteurs. Il ne faut pas oublier qu'il s'agit ici de processus partageant un contexte d'exécution et que donc l'utilisation de sémaphores et de moniteurs implantés dans la mémoire commune est possible. De même, la communication par messages entre processus

légers est facilement implantée grâce à des boîtes aux lettres appartenant au contexte.



Objet Actif Multiprogrammé

Les "*gardians*" de Argus et les "*eject*" de Eden correspondent au schéma précédent. On y trouve des objets communicant par messages. Les objets renferment de multiples activités ("*threads*") prenant en charge les opérations à effectuer. Ces activités se synchronisent à l'intérieur de l'objet soit par sémaphores (Argus) soit par moniteurs (Eden).

Le système MACH et CHORUS dans sa version V3 [ROZ88] présentent le même schéma respectivement au niveau de la tâche ("*task*") et de l'acteur : une tâche ou un acteur est un contexte d'exécution dans lequel peuvent être couplées des pages de mémoire virtuelle représentant les objets du contexte. A l'intérieur d'une tâche évoluent des activités ("*thread*") qui ont toutes accès aux objets du contexte. Une activité déclenche une opération sur un objet en envoyant un message à une activité responsable de l'exécution de cette opération (au travers de portes pour CHORUS). Un mécanisme similaire d'envoi de message permet le déclenchement d'opération dans des contextes différents éventuellement situés sur des sites distants.

Dans ces systèmes utilisant cette approche apparaît donc deux types de coopération : une coopération entre objets (utilisant l'envoi de messages au niveau interprocessus) et une coopération à l'intérieur des objets (soit sur une base de techniques classiques de synchronisation, soit sur une base de communication par messages entre processus légers). Ce double schéma augmente la complexité du noyau qui doit prendre en charge le modèle et en conséquence augmente la difficulté de sa réalisation. En conséquence aussi, la tâche du programmeur n'est pas facilitée par le nécessaire double jeu de primitives correspondant aux niveaux inter et intra objets. On a ici un défaut d'uniformité des mécanismes proposés.

Cette "*non uniformité*" se justifie par la bonne exploitation du parallélisme qu'offre

l'approche Processus Objets multiprogrammés : Si certaines fonctionnalités de l'objet peuvent être réalisées de façon concurrente, les processus légers qui les prennent en charge peuvent s'exécuter en parallèle. On a donc ici un parallélisme interne aux objets en plus du parallélisme possible entre les objets. Indiscutablement la techniques décrite ici fournit une réponse à la demande de performances. Le prix à payer est une complexité accrue dans la création des exécutifs supportant ce modèle.

OMPHALE se place dans l'approche Processus Objets monoprogrammés sous la forme du Modèle Serveur. Par rapport à la remarque précédente, nous voulons mettre en avant la simplicité et l'uniformité du Modèle Serveur.

1.4.3 Le Modèle Serveur

Le **Modèle Serveur** est un schéma d'exécution construit sur une approche par **Processus Objets monoprogrammés.** Dans ce modèle, l'entité de structuration unique du système et des applications qu'il supporte est le module Serveur,(que nous appellerons par la suite plus simplement Serveur ou Module) qui est à la fois un objet (consistance logique, interface d'utilisation, encapsulation d'informations) et un processus dont le contexte d'exécution est limité à l'objet.

On peut de suite faire la remarque qu'ici, chaque nouvelle allocation d'un processus à un processeur va entraîner un changement de contexte. Cet aspect est naturellement pénalisant en terme de performances par rapport aux mécanismes qui utilisent le partage de contexte entre processus légers. Face à cet handicap (qui peut être partiellement levé, voir le dernier chapitre sur l'architecture spécifique Site 0 à changement de contexte rapide), le modèle Serveur apporte une grande uniformité du mécanisme de coopération des entités du systèmes et de ses applications. Cela semble une approche intéressante pour la prise en charge directe des langages de programmation par modules ou objets. Jean-Marie Place développe actuellement un préprocesseur Modula-2 [WIR83] qui permet simplement d'utiliser les modules du langage comme modules Serveurs au dessus du noyau NERF [PLA88]. Ce type d'avantages liés à la simplicité et l'uniformité du modèle justifie notre étude malgré certains aspects actuellement non-performants.

La coopération entre modules Serveurs se fonde uniquement sur l'envoi de messages entre processus autonomes.

Dans le modèle Serveur, l'envoi d'un message à un module Serveur est vu comme une requête d'un service du module Serveur. Une requête précise un module Serveur, l'opération demandée et les éventuels arguments à fournir à l'opération. Lorsqu'un Serveur prend en charge une requête, il exécute l'opération demandée, on dit qu'il traite la requête.

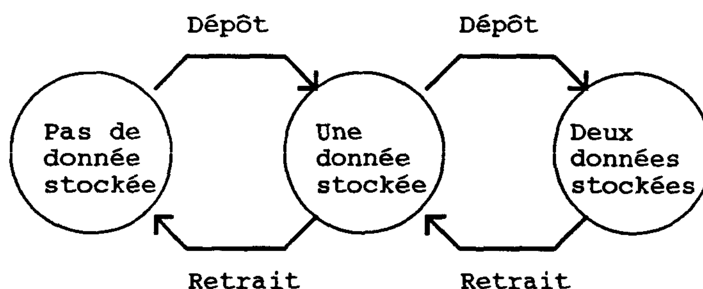
Toutes les requêtes reçus par un serveur vont être traitées séquentiellement selon

une stratégie que nous détaillons maintenant.

Un Serveur est en effet le lieu où l'on trouvera toute la gestion permettant son partage entre ses clients (ceux qui émettent des requêtes). Fondamentalement un serveur doit assurer le maintien de la cohérence de son état interne. Pour cela un certain ordonnancement du traitement des requêtes de service doit être imposé. Une autre manière de dire cela est de dire que la notion de requête de service joue le rôle de synchronisation dans le modèle , et donc, l'ordonnancement des traitements des requêtes impose la synchronisation des clients du Serveur.

Prenons un exemple simple d'un serveur d'un tampon de données. Des processus peuvent émettre vers ce serveur des requêtes pour y déposer ou y retirer des informations. Dans ce cas le serveur ne traitera pas les requêtes de dépôt si le tampon est plein (et réciproquement les requêtes de retrait si le tampon est vide). Ainsi donc le Serveur assure la cohérence des opérations effectuées vis–à–vis de l'état du tampon, et aussi crée une synchronisation de type producteur consommateur entre les processus clients. On remarquera que ce type de coopération ne demande pas d'outils de synchronisation spécifiques ; les processus synchronisés ne savent même pas que cette synchronisation a lieu : ils ne font que faire des requêtes à un serveur.

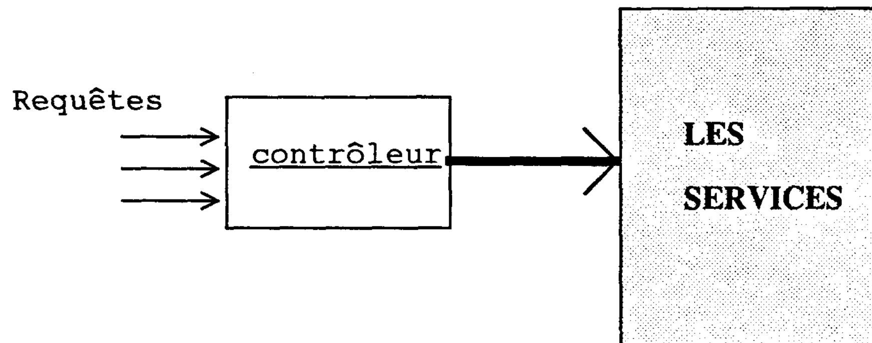
En se limitant a un tampon de deux places, l'ordonnancement des requêtes de services pour ce serveur serait ici réalisé à partir de l'automate suivant qui précise pour chaque état possible du tampon quelles sont les requêtes que l'on peut traiter pour atteindre un nouvel état possible.



Enchaînements possibles des requêtes

Ce travail est effectué par ce que nous appelons le contrôleur du Serveur. Il apparaît comme un dispositif frontal au Serveur, capable de choisir, parmi les messages reçus, un qui puisse être traité en regard de l'état du Serveur. Par rapport au modèle Objet, ce contrôleur est le gérant de l'immersion de l'objet au niveau processus: Il gère les arrivées asynchrones de messages provenant des

autres objets actifs.



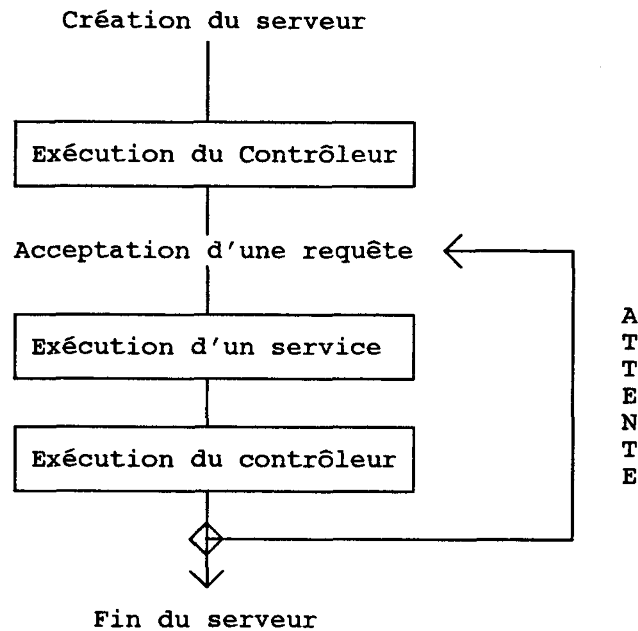
Module Serveur

Dans un processus serveur, on doit donc normalement reconnaître les composants suivants:

1) **Les services**: Un service est un composant exécutable associé à un type de requête. L'exécution d'un service correspond au traitement d'une requête.

2) **Le contrôleur** : Ce composant exécutable doit assurer un séquencement de l'exécution des services conforme au comportement voulu du module vis-à-vis des sollicitations extérieures. C'est donc lui qui effectue la sélection des traitements de service possibles.

L'exécution de ces différents composants est séquentielle et suit pour un serveur le schéma suivant:



Le langage ADA peut servir à illustrer une réalisation de ce Modèle. En ADA c'est la notion de tâches qui permet de construire des serveurs. On peut encapsuler des informations dans une tâche ADA et sa spécification contient les différentes requêtes possibles (ENTRY). L'instruction ACCEPT correspond alors à l'attente d'une requête et le corps de l'ACCEPT correspond au traitement de la requête. L'instruction SELECT WHEN est la manière ADA de réaliser le contrôleur du serveur; prise dans une boucle cette instruction permet d'ordonnancer les traitements.

```
TASK stockage_chaine IS
            -- ce serveur offre un service de stockage d'une
            -- chaine et un service de recuperation de la chaine.
            -- Un service particulier (retirer_majuscule) permet
            -- d'attendre la présence d'une chaine en majuscule.
   ENTRY deposer (ch : IN STRING);
   ENTRY retirer_majuscule (ch : OUT STRING);
   ENTRY retirer (ch : OUT STRING);
 END stockage_chaine;

 TASK BODY stockage_chaine IS
   -------------
   --- L'etat ---
   -------------
   s : STRING(80);    --chaine tampon

   -------------------
   --- Les services ---
   -------------------
   PROCEDURE service_deposer(ch : IN STRING) IS
     s(1..ch'LENGTH) := ch;
   END service_deposer;
   PROCEDURE service_retirer(ch : OUT STRING) IS
     ch(1..s'LENGTH) := s;
   END service_deposer;

   --------------------
   --.- Le controleur ---
   --------------------
   LOOP
     -- La boucle oblige l'alternance deposer/retirer
     ACCEPT deposer (ch : IN STRING) DO
        service_deposer(ch);
     END deposer;
     SELECT
       -- les demandes majuscule ne sont servies que si
       -- la phrase stockée est en majuscule.
       WHEN majuscule(s) DO
         ACCEPT retirer_majuscule (ch : OUT STRING) DO
           service_retirer(ch);
         END retirer_majuscule;
       OR
         ACCEPT retirer (ch : OUT STRING) DO
           service_retirer(ch);
         END retirer;
     END SELECT;
   END LOOP;

 END stockage_chaine;
```

Dans le Modèle Serveur, le travail du contrôleur se fonde i) sur les opérations demandées (c'est-à-dire les messages reçus) et ii) sur l'état du Serveur. Cette remarque provient de la règle générale qu'un serveur est une entité autonome, elle contient donc toutes les informations qui participent à son travail propre.

La stratégie générale du contrôleur est la suivante (nous détaillerons ces points dans le chapitre suivant qui décrit un support d'exécution du Modèle Serveur qu'est le noyau NERF):

Si aucune opération demandée ne correspond à une transition (au sens de l'automate précédent) acceptable de l'état du Serveur, le contrôleur ne déclenche aucune opération et attend l'arrivée d'autres messages qui débloqueront la situation. Dans le cas où une ou plusieurs opérations demandées peuvent être traitées, le modèle précise simplement que l'une de ces opérations sera traitée. Le choix du contrôleur peut être ici dicté par l'heure d'arrivée ou par des priorités ou encore de façon non déterministe.

On verra dans le chapitre suivant qu'un mécanisme d'"échéance" permet de libérer un contrôleur en attente et d'associer un service à l'arrivée de l'échéance. Ce mécanisme est bien connu comme étant à la base du traitement des cas exceptionnels.

1.4.3.1 Critère de disponibilité

Dans le Modèle Objet, les activités de développement consistent le plus souvent à créer des classes d'objets réutilisables, c'est-à-dire indépendamment des applications qui les utiliseront. Le travail consiste donc à enrichir progressivement et continuellement l'environnement de la machine [MEY88].

Dans le domaine des processus, on voit souvent un processus comme une tâche effectuant un travail unique pour une application donnée. Le Modèle Serveur permet de retrouver au niveau processus la réutilisabilité du Modèle Objet. Elle apparaît ici comme une disponibilité constante des Processus Objets aux applications qui désirent les utiliser. Pour cela un serveur doit être écrit en respectant un "*critère de disponibilité*" qui dit qu'un serveur ne doit pas être écrit pour un client donné mais être disponible à tous ceux qui respectent son interface (Le client doit bien sûr avoir des droits (voir la protection) suffisants pour utiliser le serveur).

Ce critère a un impact important au niveau de la couche inter-processus : Un serveur ne connaîtra pas explicitement les clients qu'il sert (c'est-à-dire être programmés pour des clients explicitement nommés), mais au contraire le serveur, lorsque cela est utile à la coopération, acquérira dynamiquement les noms des clients pour lesquels il est en cours de service. Il est donc nécessaire pour utiliser le Modèle Serveur que les mécanismes de communications inter-processus permettent le transfert d'informations de nommage des processus.

C'est une caractéristique du Modèle Serveur de définir un espace de nommage des Serveurs, espace utilisable dans les communications entre Serveurs.

On verra dans la partie nommage que nous utilisons un mécanisme classique de désignation des Serveurs par l'intermédiaire de noms dit "*locaux*" à chaque Serveur; ce qui nous permet d'ajouter une dimension de **protection** à l'espace de nommage.

Une autre technique couramment employée est celle utilisant des noms symboliques (en général par l'intermédiaire d'un serveur de fichiers). Deux

processus quelconques peuvent alors dialoguer en établissant la connexion à l'aide de noms symboliques. Cette technique est surtout avantageuse lorsqu'un processus souhaite dialoguer avec une ressource générale du système (l'accès à une imprimante par exemple). Inversement lorsque dans une application complexe il est nécessaire d'établir dynamiquement des "*liens*" entre processus, le passage obligé par des noms symboliques paraît coûteux et injustifié.

1.5 Conclusion sur les Objets Actifs

A partir du Modèle Objet, et en voulant préserver son uniformité, nous arrivons donc à un modèle d'objets actifs dit Modèle Serveur permettant la prise en compte du parallélisme entre objets. Ce Modèle se fonde sur l'équation un objet = un processus et organise la coopération des objets par l'envoi de requêtes de service (messages) au niveau inter-processus. Ce dernier point impose que chaque objet soit muni d'un contrôleur qui règle les arrivées asynchrones de requêtes de service. Pour préserver les aspects méthodologiques de l'approche objet, il faut qu'un espace de nommage des serveurs soit disponible au niveau de la coopération d'objets, c'est-à-dire dans les communications.

Une description d'un exécutif support au Modèle Serveur est donné au chapitre suivant: c'est le noyau NERF du système réparti OMPHALE. La deuxième partie de ce chapitre traite donc de la répartition et, à partir de l'étude des systèmes répartis existants, nous montrons que le Modèle Serveur peut être utilisé dans le cadre de la répartition. Les problèmes abordés sont ceux 1) des modes de transport de messages dans les architectures réparties 2) du comportement des processus vis-à-vis des communications 3) des aspects nommage et protection dans ces systèmes.

1.6 Les systèmes répartis

Notre objectif est d'introduire, grâce au modèle Serveur, le parallélisme et la répartition dans le modèle objet en vue de la création d'un système d'exploitation réparti. Un travail important a été l'analyse des solutions retenues dans la conception des systèmes d'exploitation [COR81] [TAN81b] [TAN85] [TAN87] [KRA87] pour l'exploitation de la distribution et de justifier la conservation ou l'abandon de ces solutions pour le modèle Serveur.

Nos investigations ont portées sur:

1) Les systèmes de processus coopérants et plus particulièrement le systèmes de processus communicants, tels CHORUS V2, PERSEUS [ZWA84], ACCENT [RAS85] [FIT86], AMOEBA [TAN81a] [MUL86] [REN88], et V_KERNEL [CHE83] [CHE84] [CHE84b] [CHE85] [CHE88].

2) Les mécanismes primitifs de communications inter-processus adéquats dans de

tels systèmes. C'est essentiellement la notion de transfert d'informations entre processus par message qui nous a intéressé pour son adéquation avec le modèle Serveur.

3) Le nommage à l'intérieur du système (c'est-à-dire comment les processus se désignent mutuellement), et
4) Les mécanismes de protection (c'est à dire comment protéger contre eux-mêmes de tels systèmes).

Cette partie comprend d'abord une partie introduction sur les architectures réparties (on lira aussi la synthèse de [KRA87]), leurs avantages et problèmes, puis les quatre points précédents sont discutés en citant les systèmes existants et en les éclairant à la lumière de l'expérience OMPHALE donc du Modèle Serveur.

1.7 Architectures réparties

1.7.1 Introduction

Les progrès des techniques des supports de communication permettent maintenant la création effective d'architectures matérielles réparties. On entend par là ensemble de **sites** (c'est-à-dire structures matérielles quelconques mono ou multi-processeurs) reliés par un support de communication grande vitesse à fiabilité élevée (un exemple très répandu d'un tel support de communication est ETHERNET [MET76]).

On exclut du terme réparti les architectures matérielles dites parallèles formées de multiples processeurs reliés par un ou plusieurs bus (machines multiprocesseurs à mémoire commune, machines systoliques ou hyper-cubiques). Néanmoins une telle architecture peut très bien être un site d'une architecture répartie. En effet, comme la plupart des systèmes existants, le noyau NERF d'OMPHALE crée sur chaque site une machine virtuelle OMPHALE qui cache les spécificités de l'architecture physique.

Avec les architectures réparties, l'utilisateur doit profiter des potentialités nouvelles liées à la répartition. Les équipes de recherches en Conception de Système d'Exploitation travaillent à la conception de nouveaux systèmes d'exploitation dits **répartis**. Le but de la plupart de ces travaux est de fournir à l'utilisateur une **ARCHITECTURE DE SYSTEME** virtuelle cachant tous les problèmes de gestion de la répartition matérielle, offrant ainsi à l'utilisateur l'apparence d'une machine centralisée alors qu'elle est physiquement et logiquement éclatée.

1.7.2 Avantages et Problèmes

Examinons les avantages et problèmes d'un système réparti.

Les avantages immédiatement perceptibles d'un système réparti sont les suivants:

Régulation de charge :

La charge d'un système **centralisé** est celle du seul site existant. Sur un système multi-site, un utilisateur va pouvoir travailler (dans l'idéal de façon totalement transparente) sur un site distant si le site local est trop chargé. L'utilisateur peut espérer des performances constantes grâce à une **régulation automatique de charge** fondée sur des mécanismes de base d'exécution à distance ou de migrations de tâches.

Partage des ressources :

Sur un système réparti, les ressources gourmandes (en temps: gros logiciels, en matériels: périphériques coûteux) peuvent être associées à un site sans qu'il soit nécessaire de les dupliquer sur chaque site. Ce partage doit être, là aussi, géré par le système de façon transparente aux utilisateurs. Lors de l'utilisation d'application importantes, on gagne aussi en évitant le transfert (par le réseau) de l'application vers le site de l'utilisateur.

Parallélisme :

Le parallélisme physique obtenu par la présence des différents sites est évidemment une donnée de base des systèmes répartis. Le système se doit d'en tirer toutes les possibilités. Les nouveaux schémas d'exécution (exécution parallèle, processus coopérants) peuvent être utilisés sur ces architectures.

Disponibilité élevée :

Sur un système centralisé, la panne du site central bloque toutes les tâches. Dans un système réparti la panne d'un site ne bloque pas automatiquement tout le système (et tous ses utilisateurs). Les tâches n'utilisant pas les ressources du site en panne peuvent continuer à travailler. Pour les autres tâches, qui sont soit bloquées par l'absence de ressources nécessaires ou tuées par la panne du site, des possibilités de reconfigurations complètes (nouveau site) ou partielles peuvent exister.

Ces nouvelles architectures ont à contrario fait naître des **problèmes nouveaux**:

Absence de mémoire commune :

Dans un système centralisé, toutes les informations sur l'état du système et des applications se trouvent dans la mémoire du site, il est donc possible d'évaluer un état instantané du système. Dans un système réparti, les informations d'état sont *physiquement* distribuées dans les mémoires des sites et modifiées de façon asynchrone et simultanée par les différents processus. En conséquence, l'état de

l'ensemble de système ne peut être connu avec exactitude à un instant t par un processus donné. La gestion du système ne peut donc se faire qu'à partir d'informations qui ne sont pas obligatoirement mises à jour. Cet aspect important des systèmes répartis se retrouve aussi effectivement au niveau de l'écriture d'applications sur ces systèmes et des systèmes eux-mêmes. Les différents composants d'une application ne doivent pas présupposer l'existence d'une mémoire globale à tous les composants.

Transparence de la répartition :

Cette notion implique que l'accès à une ressource distante doit se faire de la même façon que l'accès à une ressource locale. Cela peut être réalisé par la recherche d'un système de désignation (nommage) unique étendu à l'ensemble des sites.

Régulation de charge :

Pour efficacement utiliser le présence de différents sites, le système doit répartir la charge globale sur l'ensemble des sites. Cela s'appelle de la **régulation de charge**. Pour pouvoir réaliser cette fonctionnalité, le système doit offrir les mécanismes de base adéquats (comme par exemple la diffusion pour permettre la communication d'informations de charges sur tous les sites, ou la migration de tâches pour pouvoir adapter la charge).

Création d'une Machine virtuelle :

Lorsque le système doit apparaître comme une système unique alors qu'il est matériellement et logiquement réparti, il est nécessaire que ce système possède des fonctionnalités internes qui lui permette d'offrir cette apparence. Cela est un problème difficile car tous les problèmes liés à la répartition doivent être résolus pour la machine virtuelle.

Gestion du parallélisme :

Le parallélisme inhérent à l'architecture entraîne une demande d'outils nouveaux pour l'exploiter. Tous les modèles d'exécutions parallèles décrits pour les systèmes centralisés ne sont pas ou difficilement utilisables en univers réparti.

Détection et Tolérance aux pannes :

Sur un système réparti, la panne d'un site ou la perte d'informations lors de la communication d'informations sur le réseau sont des événements qui n'existent pas sur les systèmes centralisés. Vu des sites actifs , de tels événements correspondent à un mauvais fonctionnement de composants (logiciels et matériels). Ces mauvais fonctionnements doivent être détectés et si possible corrigés. Cela participe à ce qui s'appelle la **tolérance aux pannes.**

De tous ces points, le projet OMPHALE, par son modèle Serveur, s'intéresse à

l'exploitation du parallélisme et de la distribution. La structuration du système se fait grâce à des objets actifs qui sont à la base du partage des ressources sur une architecture sans mémoire commune. Les noyaux NERF d'OMPHALE créent une machine virtuelle qui rend transparent la distribution des composants du système et des applications.

OMPHALE , pour des raisons de simplicité, ne fait pas actuellement de migration de modules serveurs, ni donc de régulation de charge.

OMPHALE n'introduit pas de mécanismes particuliers pour la tolérance aux pannes, des solutions classiques devront être utilisées au niveau de la couche CORTEX (voir plus loin) en particulier par la Station de Transport qui encapsule le réseau physique, et au niveau utilisateur par une programmation défensive.

1.7.3 Création d'un système réparti

La création d'un système réparti peut se concevoir comme :

L'assemblage de systèmes classiques sur les différents sites grâce à une couche logicielle qui filtre les commandes pour les sites distants et utilise le réseau pour les réaliser . L'exemple en est NFS de Sun [SAN86] qui permet l'assemblage de sites UNIX ou autres: ici le gestionnaire de fichiers a été étendu à l'ensemble des sites, l'utilisateur voit les systèmes de fichiers des sites comme un ensemble d'arborescences. Déplacer un fichier dans ces arborescences va entraîner une éventuelle migration du fichier de façon transparente à l'utilisateur. Un autre exemple est PCTE–Emeraude [ESP86] qui substitue aux gestionnaires de fichiers de sites UNIX un Système de Gestion d'Objets étendu à l'ensemble des sites.
L'intérêt d'une telle méthode est bien sûr d'utiliser des systèmes existants qui ne sont pas à récrire. L'inconvénient est que les systèmes sous–jacents n'ont pas été prévus pour la répartition et bien souvent n'offrent pas les outils nécessaires à la construction d'une couche répartie efficace et sûre.

La création d'un système adapté au besoin de la répartition est une solution qui permet, dès les spécifications initiales, de prendre en compte les besoins propres des systèmes répartis (communications entre composants sans mémoire commune, mode de désignation unique, régulation de charge, tolérance aux pannes). La plupart des systèmes novateurs sont de ce type.

La tendance actuelle de conception des systèmes répartis est en une décomposition à deux niveaux. Un noyau réduit dupliqué sur chaque site et une couche système répartie au dessus. Tous les systèmes ici cités utilisent ce découpage. OMPHALE reprend cette décomposition. Détaillons ces deux couches.

1) Un **Noyau réduit** offrant un ensemble restreint de primitives. Ce noyau est reproduit sur les différents sites. Il réalise en général les fonctions primitives d'allocation des processus au processeur, la récupération des interruptions, la prise

en charge des communications.

Les avantages de la création du système à partir d'un noyau réduit sont de deux ordres : i) Ce noyau crée, sur les machines où il est installé, une machine virtuelle prête à supporter le système voulu, c'est une bonne approche des problèmes de portabilité et d'hétérogénéité, ii) Le noyau servant à réaliser les mécanismes de base du système créé, il est intéressant de pouvoir développer la plus grande partie du système à l'aide de ces mécanismes de base. Cela est d'autant plus vrai que le noyau réalise une machine virtuelle aux fonctionnalités novatrices.

Le choix des mécanismes de base à mettre dans un tel noyau réduit est souvent fait pragmatiquement à partir de critères de performances. En effet la création d'une machine virtuelle entraîne en général une perte d'efficacité par rapport à une utilisation directe de la machine physique. Il faut ici réaliser un compromis entre cette perte d'efficacité et les avantages obtenus. Les fonctionnalités qui se retrouvent le plus souvent dans ces noyaux réduits sont i) la gestion des processus locaux au site et l'allocation de ces processus aux processeurs (ordonnancement de bas niveau), on trouve ici aussi la réalisation d'un mécanisme de nommage étendu au système complet ii) la mise en oeuvre des communications locales entre processus iii) le traitement élémentaire des interactions avec la périphérie. Pour d'autres fonctions de base, le choix paraît plus discuté : i) Lorsque la protection est réalise par "**capacités**", ces capacités sont souvent elles-mêmes protégées dans le noyau. AMOEBA, cependant, les place au niveau utilisateur en les cryptant. ii) Le mécanisme de transport de messages entre site est souvent "hors noyau", AMOEBA, la aussi, se distingue en intégrant au noyau le gestion du réseau physique ETHERNET, et cela pour pouvoir identifier les messages AMOEBA et les trames ETHERNET. C'est ici un réel souci d'efficacité.

2) Une **Couche Système** implantant les fonctionnalités du système. Cette couche est répartie sur l'ensemble des sites. Cette couche est elle-même décomposée fonctionnellement en sous-systèmes relativement autonomes les uns par rapport aux autres. On pourra trouver un gestionnaire de mémoire, un gestionnaire de fichiers, le sous-système de communication inter-processus, les processus d'entrée sortie, le sous-système assurant la protection etc...

Cette couche système est développée sur la machine virtuelle formée des noyaux des sites et d'un système de transport de messages inter-sites. Le développement de cette couche bénéfice donc des concepts (nouveaux) implantés dans la machine virtuelle.

Un sous-système, formé d'un ensemble de composants peut être situés entièrement sur un site dit spécialisé, et utilisable à distance, ou être réparti sur l'ensemble des sites, et utilisable à partir des composants locaux. Toute solution intermédiaire entre ces deux extrêmes peut exister. Par exemple, dans beaucoup de systèmes, le système de fichiers est réparti , citons CHORUS: chaque site muni d'une mémoire de masse logique assure sa gestion, ces différents sites coopèrent pour construire virtuellement un seul gestionnaire de fichiers. Le V-KERNEL, au

contraire, utilise un serveur de fichiers unique installé sur un seul site, son utilisation se fait uniquement à distance au travers du réseau. Les recherches de V–KERNEL pour un transfert de message très efficace, l'utilisation d'"*équipes*" de processus, la présence de mémoires caches ont permis d'obtenir des performances qui valident ce choix, voir [CHE83].

Le système GUIDE utilise lui aussi des objets répartis appelés "*domaines*". Un domaine matérialise un contexte d'exécution et regroupe un ensemble d'objets. Un domaine peut s'étendre sur plusieurs sites.

Le système OMPHALE utilise cette décomposition en deux niveaux:

1) Un noyau appelé **NERF** (Noyau d'Exécutif Réparti) reproduit sur chaque site de l'architecture et qui réalise un **Site Virtuel OMPHALE** capable de prendre en charge les processus OMPHALE du site ainsi que leurs communications. Il réalise aussi l'interface avec le matériel en récupérant les sollicitations périphériques.

2) Au dessus de l'ensemble des NERFs, une couche système appelée **CORTEX** (Collection d'Outils RéparTis pour l'EXécution) formée d'un ensemble de modules serveurs réalisant les principales fonctionnalités d'un système d'exploitation : Serveur de Fichiers, Serveur de Mémoire, Serveur de Noms Symboliques, Gérants de Périphériques, Station de Transport... Certains de ces serveurs sont obligatoirement présents sur chaque site, tel le Serveur de Mémoire; d'autres ne peuvent être présents que sur des sites dédiés, tels les Serveurs de Fichiers associés aux mémoires de masse. Une Station de Transport qui réalise le routage des messages est formée d'un ensemble de processus coopérant réparti sur l'ensemble des sites.

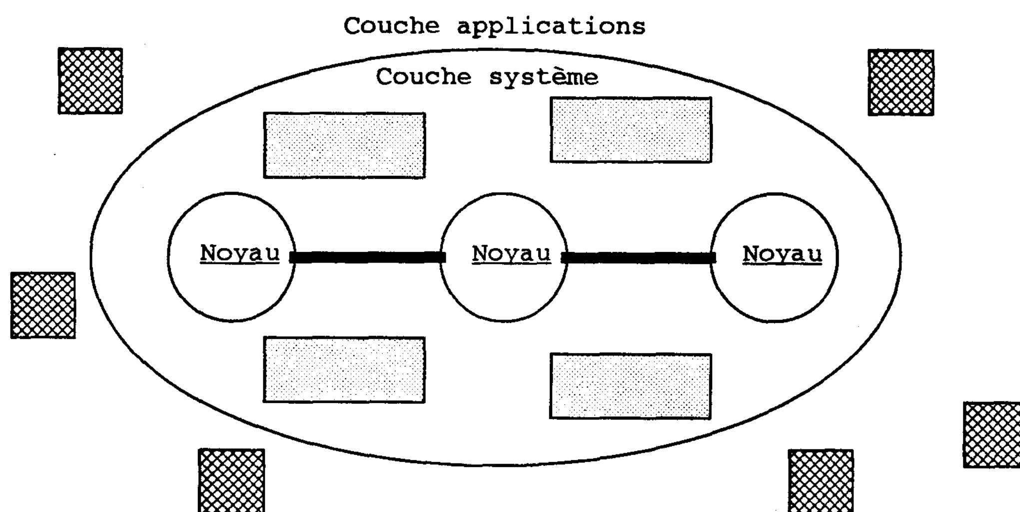
Au dessus on trouve la couche application formée des modules utilisateurs. Cette couche est logiquement répartie: à ce niveau la transparence de la répartition est assurée par l'architecture.

A ce niveau la localisation des modules utilisateurs est quelconque. **Un module est entièrement situé sur un site**, mais un ensemble de modules coopérant à la réalisation d'une application ou d'un sous–système peut être réparti sur l'ensemble des sites.

Avec le modèle Serveur, le contexte d'exécution d'un processus est strictement limité à l'objet qu'il supporte. Il n'est donc pas nécessaire d'avoir de la répartition à un niveau sub–objet. Dans d'autres systèmes, le contexte d'exécution d'un processus dans un objet va référencer un ensemble d'objets, il est alors intéressant d'avoir des possibilités de répartition à ce niveau. C'est le cas du système GUIDE et de ses objets "*domaines*" qui matérialisent des contextes d'exécution. C'est aussi le cas du système SOS dans lequel un contexte d'exécution renferme des "*mandataires*" d'objets éventuellement distants. Ici ce n'est pas le contexte d'exécution qui est réparti, mais c'est l'objet et ses mandataires. Pour le processus, l'accès à un objet se fait par simple appel d'une

opération d'un mandataire. La complexité de l'objet réparti est ainsi cachée au client, en particulier, pour le processus l'objet paraît local et unique. Ainsi donc: *"Dans SOS, ce n'est pas le client qui va au service (par messages), c'est le service qui va au client (en lui donnant un mandataire)"*.

La présente thèse se consacre au noyau NERF d'OMPHALE, pour lequel a été réalisé une plate-forme expérimentale au dessus du noyau temps réel SPART des machines Unix SPS7 de BULL. Cette plate-forme doit permettre l'étude de la construction de la couche CORTEX. C'est le travail actuel de Jean-François Méhaut. Les solutions retenus pour la couche CORTEX, encore à l'étude, ne seront donc que citées en fin du chapitre consacré à la description de NERF.



Sructuration d'un système réparti

1.7.4 Primitives de communication

Dans un système de processus, toute application est réalisée par la **coopération** de processus. Cette coopération est réalisée en termes de **synchronisation** et de **communication.**

La synchronisation règle de déroulement des différents processus dans le respect de règles telle que l'accès exclusif à des sections critiques etc... Lorsque deux processus doivent se synchroniser, il s'agit le plus souvent d'un besoin pour ces deux processus de partager de l'information (accès à des données communes, travail en producteur consommateur ...) . On dit alors que la communication est construite sur la synchronisation.

Ce type de technique est généralement utilisée en univers centralisé: les processus se synchronisent pour l'accès à une structure commune (tel un tampon) pour réaliser la communication. Différentes techniques ont été introduites, on peut citer les sémaphores (de bas niveau) et les moniteurs de Hoare (de plus grande lisibilité). Elle permettent d'exprimer les besoins en terme de synchronisation et en conséquence de communication.

En univers réparti, ces techniques ne peuvent être directement utilisées vu l'absence physique de mémoire commune [LIS79]. Elles peuvent cependant être retrouvée dans une approche objets ("classique" au sens de la section précédente). Ici le système assure un protocole d'accès aux objets, uniforme et étendu à l'ensemble du système réparti et les processus se déroulent en sautant d'objets en objets et se synchronisent dans des objets partagés. La synchronisation utilise alors des mécanismes classiques qui sont le reflet des techniques centralisées mais ici limités à une entité passive plus fine : l'objet. On retrouve les mêmes techniques de synchronisation classiques dans une approche Processus Objets multiprogrammés.

Une deuxième approche des problèmes de synchronisation et de communication consiste à dire qu'il n'y a besoin de synchronisation que pour réaliser la communication , et que donc en conséquence, un système fondé sur des mécanismes puissants de communication réglera par là tout problème de synchronisation sans introduction de mécanismes particuliers. Cette approche est à la base des l'ensemble des systèmes à processus communicants dont OMPHALE fait partie et dans lequel on retrouve les systèmes CHORUS, ACCENT, AMOEBA, PERSEUS et V-KERNEL.

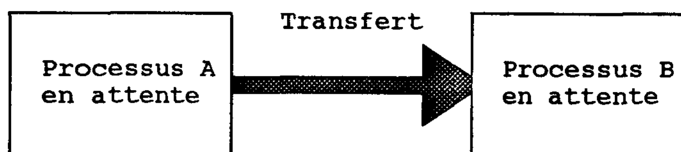
Dans ces systèmes, le schéma d'exécution des processus est décrit exclusivement à partir de primitives de communication. Nous allons maintenant faire une synthèse des concepts de communication.

La communication, dans une architecture sans mémoire commune, a toujours comme but ultime de permettre le transfert d'informations d'un espace mémoire à un autre. Les différences viennent ensuite du comportement des communiquant vis-à-vis de la communication.

Deux mécanismes de base de transfert peuvent être utilisée : le **transfert synchrone** et le **transfert asynchrone**, nous les détaillons ci dessous. La communication par messages est une technique de réalisation de ces transferts, elle permet la réalisation de communications synchrones ou asynchrones. Elle est détaillée ensuite.

1.7.5 Transfert synchrone

Ici le transfert d'informations entre processus communicants exige la présence attentive des processus communicants. On dit qu'il y a attente respective des deux processus communicants à un **point de communication**, on parle de **rendez-vous.** Le transfert d'informations a lieu lorsque les deux communiquants sont en attente sur le point de communication. On parle de **transfert synchrone.** L'image naïve correspondante est celle du téléphone : pour parler à votre correspondant, vous devez et il doit décrocher!!!



Transfert synchrone sur point de communication

Les avantages de ce type de communication sont de plusieurs ordres:

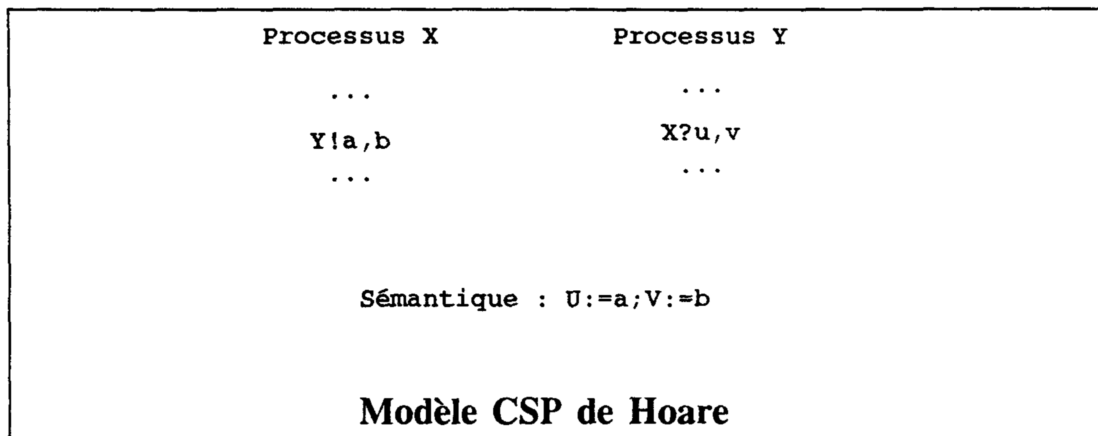
i)Le transfert d'informations d'un espace mémoire à l'autre se fait lorsque les processus communicants sont bloqués. Ce transfert peut donc être entrepris par le système sans crainte de conflits lors de l'accès aux espaces mémoires. Une conséquence immédiate est qu'il n'y a pas besoin d'un stockage intermédiaire des informations à transférer.
D'autre part, sur un point de communication, le transfert d'informations peut être éventuellement bidirectionnel avec transfert d'informations dans les deux sens au cours de la communication.

ii)Le bon ou mauvais déroulement du transfert synchrone peut se fait alors que les processus communicants sont bloqués. Les processus communicants ont donc une connaissance immédiate et précise des conséquences de la communication. Cela simplifie fortement l'établissement de points de reprise sur échec de communication car l'état des processus communicants est parfaitement défini. C'est une sémantique simple qui a le mérite de cacher derrière la communication les mécanismes éventuellement compliqués des protocoles de transport sous–jacents.

L'inconvénient de ce type de solution provient de la forte synchronisation : L'un doit attendre l'autre alors qu'il aurait peut–être eu, une fois les informations préparées, autre chose à faire. De plus, dans une architecture répartie, le transfert

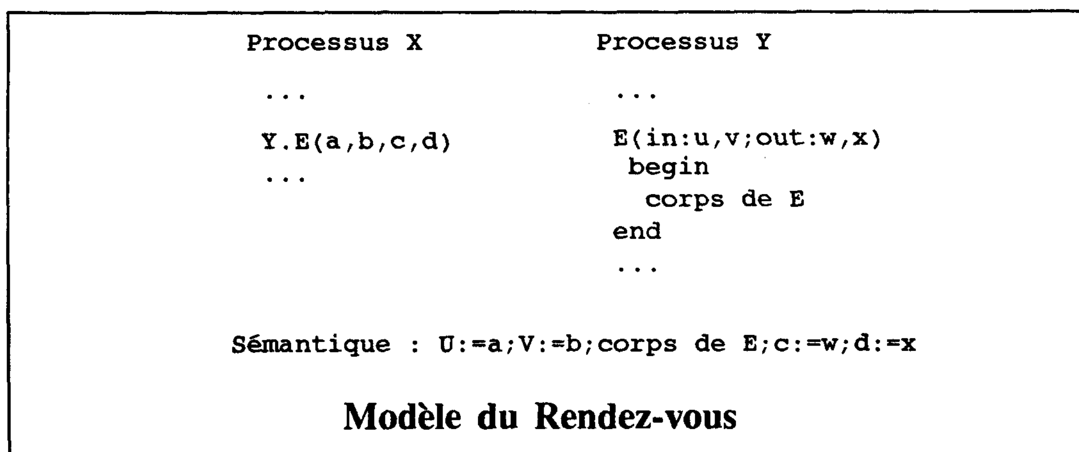
physique peut mettre en jeu un mécanisme coûteux pendant lequel les processus communicants restent bloqués.

Ce type de transfert est à la base du modèle CSP de Hoare [HOA78] et du langage OCCAM [INM84a].



Modèle CSP de Hoare

La communication sur Rendez-vous de ADA [DOD84] et le Remote Procedure Call [BIR85] sont des extensions de ce type de communication. Ici le transfert d'informations est bidirectionnel avec exécution d'un morceau de code chez l'appelé. Ce type de communication introduit une relation Maître/Esclave entre les processus communicants.



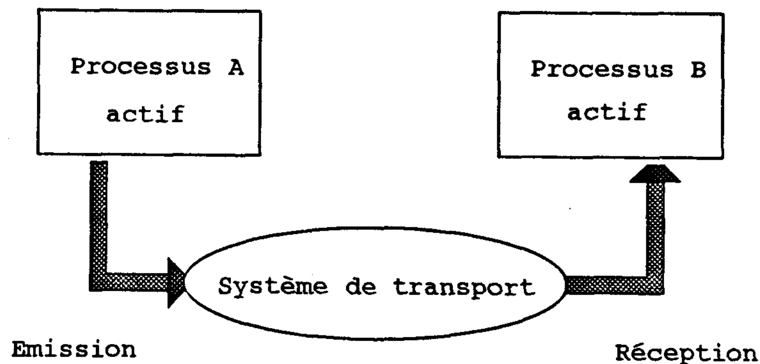
Modèle du Rendez-vous

1.7.6 Transfert asynchrone

La technique ici consiste essentiellement, par rapport à la précédente, à désynchroniser les processus communicants en permettant à l'émetteur d'informations de les envoyer dès qu'il les a préparées, et au récepteur de les recevoir dès qu'il se juge prêt. L'image naïve correspondante est le service postal des PTT : vous postez votre lettre quand vous le voulez, je regarde ma boîte aux lettres quand je le veux. Bien évidemment le système doit organiser un **Système de Transport** permettant d'acheminer les informations d'un processus à l'autre. Ce Système de Transport doit en conséquence stocker les informations à transférer dans l'espace inter–processus. On parle de **transfert asynchrone.**

En général, avec un tel mécanisme, les processus communicants ne peuvent être garantis contre la perte d'informations lors du transfert, ni contre la non–réception des informations. Ces "*pannes*" peuvent , à la fois être difficile à détecter, et avoir des conséquences importantes et non locales sur le déroulement des processus. Difficile à détecter car, sur un système réparti, l'état global et instantané du système ne peut être connu. La solution passe par des mécanismes d'acquittement et d'échéance. Conséquences importantes et non locales, car la panne d'un composant, si elle n'est pas détectée, n'empêche pas l'application de continuer, avec des changements d'états des autres composants. Lorsque la panne est enfin détectée, il faut alors remettre l'ensemble des composants dans un état cohérent, ce qui peut s'avérer très difficile comme par exemple pour des composants partagés entre plusieurs applications. La solution passe ici par des mécanismes de transactions qui assurent que si une opération (même très complexe) échoue, le système se retrouve dans l'état précédant l'appel,

Toute ces solutions entraîne une programmation défensive des processus. Le style de programmation induit s'en trouve en général très alourdi (nombreux points de reprise). On peut supposer que les langages de programmation de haut niveau pour les architectures réparties pourront prendre en compte ces problèmes, et laisser au programmeur le soin de se concentrer sur l'application elle–même.



Transfert asynchrone

L'avantage principal du transfert asynchrone est qu'il découple fortement les entités communicantes. C'est alors une manière d'exploiter le parallélisme physique de l'architecture.

D'autre part l'asynchronisme est souvent mis en avant comme mécanisme permettant la création de **Systèmes Temps Réel** (voir CHORUS V2).

1.7.7 La communication par message

Le transfert d'informations d'un processus à un autre sur une architecture répartie se fait généralement par **messages.** Un message est le "*contenant*", l'information à transférer est le "*contenu*". Une communication utilise l'échange de messages entre les processus communicants.

Outre l'information à transférer, un message peut contenir d'autres informations telles : nom de l'émetteur, du récepteur, clefs permettant de distinguer la transaction, nom d'une procédure à déclencher chez l'appelé etc.. Ces informations de contrôle sont définies par le protocole de communication utilisé par les processus communicants. Certaines informations peuvent être imposées par le système (c'est toujours le cas du nom du récepteur) ou uniquement par accord mutuel des participants (clef de transaction par exemple).

Dans CHORUS V2, un message comprend toujours l'indication de l'émetteur et du récepteur (il s'agit en fait ici de "*portes*" émettrice et réceptrice (voir plus loin)). Dans PERSEUS, c'est le noyau du système qui place le nom (ici un "**lien**) de l'émetteur lorsque celui-ci demande un protocole question-réponse. Dans le cas de communication unidirectionnelle, le nom de l'émetteur n'apparaît pas dans le

message.

Dans CHORUS–V2, un message comporte un numéro de séquence qui identifie uniquement le message. Ce numéro de séquence peut être conservé dans un message de réponse. Ce numéro peut alors être utilisé comme identifieur de la transaction. Dans PERSEUS, un message contient toujours un numéro de canal qui sert généralement à caractériser l'opération demandée, et un numéro de code qui sert typiquement à identifier l'émetteur du message.

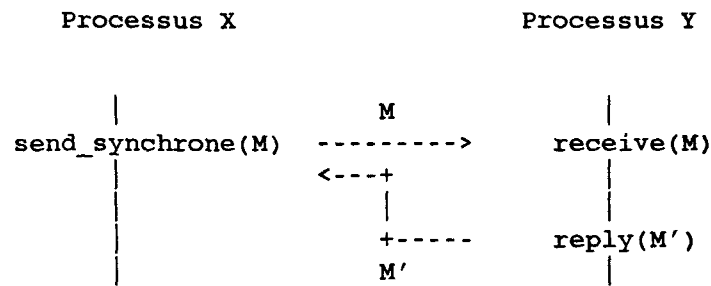
Dans notre Modèle Serveur, un message est une requête de service. Une telle requête est formée i) du nom du serveur destinataire, ii) du service demandé, iii) du contenu du message. Toute autre forme de coopération doit être construite au dessus et donc hors noyau. S'il peut apparaître coûteux en écriture de ne se servir que de la requête de service, la simplicité qu'elle apporte et l'aide certaine de langages adaptés doivent permettre de valider un tel format de message.

Dans des systèmes sans mémoire commune, les seules informations qui peuvent apparaître dans un message sont des **valeurs**. Certains systèmes permettent le passage contrôlé d'**adresses** au sens adresse mémoire (il s'agit alors de permettre l'utilisation directe dans un espace mémoire d'informations de volume important qu'il serait pénalisant de transférer physiquement (voir PERSEUS et V–KERNEL plus loin). Souvent on trouvera aussi des **noms** dans les messages. Il s'agit alors du transfert d'informations décrivant l'espace inter–processus (voir la section sur le nommage).

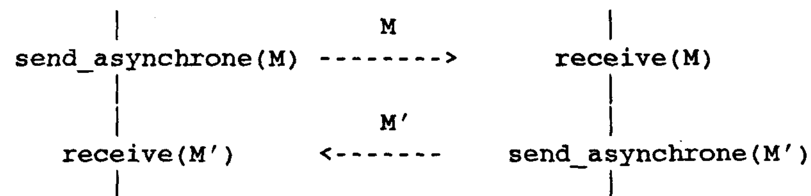
L'utilisation de messages est un mécanisme primitif de transfert d'informations. Il peut servir à réaliser différents types de communication.

Communication synchrone :
L'envoi d'un message est "*bloquant*". Le processus émetteur est bloqué jusqu'à la réception d'un message de réponse. Il y a donc ici nécessairement un transfert bidirectionnel de messages.
Du côté du récepteur, la réception du message peut entraîner l'envoi immédiat d'un message de réponse qui est ici un simple acquittement de la réception (on retrouve ici l'implantation par messages du point de communication de OCCAM). Ou bien la réception du message entraîne l'exécution d'un travail chez le récepteur qui enverra un message de réponse ultérieurement (on retrouve l'implantation par messages du rendez–vous ADA et des RPC).

Communication asynchrone :
L'envoi d'un message est "*non bloquant*". Le processus émetteur continue son travail après l'envoi. Il peut ou non être averti de la correcte prise en charge du message par le Système de Transport. Il s'agit ici d'un transfert unidirectionnel.
Le récepteur peut ou non envoyer un message de réponse mais il s'agit alors d'une autre communication entre les processus.



Communication synchrone



Communication asynchrone

Comme on le voit, la communication asynchrone entraîne une grande symétrie dans le dialogue entre deux processus, cela favorise la réalisation du modèle Serveur, dans lequel toute entité doit être vue comme un serveur des messages qu'il reçoit. La communication synchrone, elle, forme un couple Maître–Esclave dans le dialogue entre processus, et donc favorise la dualité entité active – entité passive contraire au modèle Serveur.

L'intérêt de la communication asynchrone par messages est aussi la plus grande souplesse qu'elle permet dans la coopération de processus. Les mécanismes suivants se construisent facilement au dessus de la communication asynchrone par messages.

1) Envoi d'Ordres : Un ordre est un message envoyé à un processus pour simplement le débloquer d'un état d'attente. C'est donc un message qui ne nécessite par de réponse.

2) Technique du Long Retour [STR81] : Avec cette technique, une réponse peut être envoyée directement du processus qui la calcule vers le processus qui l'attend sans remonter tous les processus intermédiaires qu'avait éventuellement nécessités la coopération. Avec le Remote Procedure Call, il faut "remonter" obligatoirement les appels imbriqués.

3) Technique des Continuations [COR81] : Avec cette technique, une réponse peut être envoyée à un processus tiers sans repasser par le processus qui a posé la question. Avec une communication systématiquement organisée autour du protocole question/réponse cela n'est pas possible.

Tous ces mécanismes ont par ailleurs l'avantage d'augmenter le parallélisme global d'une application (meilleure exploitation du parallélisme possible).

Dans le domaine des systèmes répartis à processus communicants, différents systèmes se fondent sur le transfert synchrone :

AMOEBA de A.Tanenbaum utilise le modèle du "*Remote Procedure Call*" (RPC) entre clients et serveurs. Les messages sont ici de taille variable jusqu'à 32Kilo–octets et ne sont pas stockés dans le noyau mais transférés d'un processus à l'autre au cours du RPC. Les avantages résident dans la simplicité et l'efficacité du noyau. Néanmoins, des messages particuliers dits EXCEPTION permettent d'éviter l'utilisation de l'appel procédural bloquant pour des cas exceptionnels. Un tel message cause une interruption chez le destinataire qui retourne alors un message REQUEST_ABORTED. Lorsque les appels procéduraux distants (RPC) sont imbriqués, un message d'exception débloque tous les processus en attente. Cela permet de clôturer proprement un RPC.

V–KERNEL de Cheriton et Zwaenepoel veut offrir un mécanisme simple et efficace de transfert synchrone. Un processus peut envoyer un message de taille fixe (32 octets) par une opération d'envoi bloquante en attente d'un message de réponse. Le message envoyé et la réponse sont stockés dans un même registre du processeur supportant l'émetteur. Il s'agit ici d'une technique dont la justification est l'amélioration de l'efficacité du transfert. La taille des messages est ici volontairement petite pour "entrer" dans un registre, mais alors cette taille ne permet pas le transfert de grande quantité d'informations. V–KERNEL propose pour cela des primitives (COPYTO et COPYFROM) qui à partir d'adresses physiques (passées dans les petits messages) permettent de faire des transfert de blocs mémoires de processus à processus. Il y a donc ici deux sémantiques de transfert pour petites (primitives SEND, RECEIVE et REPLY) ou grandes (primitives COPYTO et COPYFROM) quantités d'informations.

PERSEUS de Zwaenoepoel utilise aussi cette double sémantique. L'équipe de PERSEUS justifie cela de la manière suivante: On peut remarquer que lorsque la quantité d'information à transmettre est petite, cette information est souvent "*typée*", c'est- à–dire explicitement interprétée par le receveur, tandis que lorsque la quantité d'informations est importante, il s'agit d'informations non interprétées (comme le transfert du contenu d'un fichier par exemple).

Cette distinction a pourtant le désavantage de sa dualité: Il faut préciser deux sémantiques de transfert de messages. PERSEUS remarque qu'il lui paraît difficile de concevoir une technique de transfert unique (petite et grande quantité d'informations) tout en gardant les propriétés de **Portabilité** du système. La Portabilité (c'est–à–dire l'indépendance vis–à–vis de particularités d'une machine donnée) était une expérimentation importante des projets PERSEUS et THOTH [CHE82], elle n'est pas détaillée ici.

D'autres systèmes se sont tournés vers le transfert asynchrone :

ACCENT de Rashid et CHORUS dans sa version V2 (la version V3 utilise le RPC synchrone) sont des systèmes fondés sur la communication asynchrone entre processus, et ici par l'intermédiaire de "*portes*". Nous parlerons en détail de la notion de portes dans la section sur le nommage, elle ne nous intéresse ici que comme outil pour le stockage des messages dans la communication asynchrone. De façon simple une porte est une entité dans laquelle un processus peut déposer ou retirer des messages. Il existe des possibilités d'attente sur une ou des portes ou de diffusion de messages sur un ensemble de portes, etc.. Toutes ces possibilités permettent d'établir des coopérations évoluées entre processus coopérants.

Néanmoins la base fondamentale de la coopération entre deux processus par l'intermédiaire de portes est celle du producteur consommateur, comme l'est à un moindre niveau la notion de "*pipe*" UNIX. Une porte est la réalisation d'une file d'attente de messages entre des processus. C'est en effet une bonne justification de l'asynchronisme que de dire constater l'importance du modèle du producteur consommateur dans la structuration de systèmes complexes. Il m'apparaît que cette constatation est une justification fondamentale de la communication asynchrone à l'aide de portes.

La notion de porte ne se limite cependant pas à cet aspect, mais apporte aussi de nombreux avantages dans la réalisation du système. Jouant son rôle d'intermédiaire entre processus, la communication devient "**anonyme**" : les processus communicants ne se connaissent pas directement. Cela les découple fortement et permet ainsi, par exemple, des reconfigurations dynamiques derrière une porte (modification dynamique de la procédure de traitement des messages, changement du processus ou de l'ensemble des processus derrière une porte etc...). Toute ces possibilités apportent des possibilités importantes pour la création d'applications complexes sur la base de processus communicants.

Les systèmes PERSEUS et CHORUS permettent en fait à la fois les communications synchrones et asynchrones.

Dans CHORUS V2, un message comporte la désignation de la porte émettrice, la désignation de la porte ou du groupe récepteur, le contenu du message (séquence d'octets) et un **numéro de séquence** qui identifie uniquement le message. L'envoi de message peut être asynchrone (opération **msput**) ou synchrone (opération **mscall**). D'un point de vue de récepteur on peut utiliser les opérations **msputfwd** et **msputreply** qui permettent de "répondre" à un message sans modifier son numéro de séquence.

Trois protocoles de transfert de message existent dans PERSEUS:

1) **Question–Réponse Synchrone**: L'émetteur du message se bloque en attente d'un message de réponse.

2) **Question–Réponse Asynchrone**: L'émetteur continue après avoir envoyé le message, mais l'émetteur doit s'attendre à recevoir une réponse.

3) **Ordre Asynchrone**: L'émetteur continue après avoir envoyé le message, et n'attend pas de réponse.

Ces protocoles utilisent 6 primitives d'émission et de réception:

1) **Send**: Envoi d'un message et blocage de l'émetteur jusqu'à ce que le destinataire reçoit le message et envoie une réponse (par Reply). Le noyau génère automatiquement un "*reply–link*" sur lequel la réponse va être envoyée. Le Send peut être exécuté avec échéance pour éviter que le processus ne reste bloqué indéfiniment.

2) **ASend**: Envoi d'un message sans bloquer l'émetteur. Comme une réponse est attendue, le noyau génère un lien de réponse comme dans le Send.

3) **Remark**: Envoi d'un message sans blocage ni prévision de réponse (pas de reply–link).

4) **Receive**: Blocage du destinataire jusqu'à réception d'un message (avec éventuellement utilisation d'une échéance).

5) **AReceive**: Essai de réception de message, on continue si aucun message n'est disponible.

6) **Reply**: Envoi d'une réponse à un message précédemment reçu. Le reply–link utilisé est automatiquement détruit.

Au cours des développements autour de PERSEUS, il est apparu qu'un programmeur a tendance à utiliser le protocole Question–Réponse synchrone qui lui paraît plus familier (il s'apparente à l'appel de procédure). Les protocoles Asynchrones furent néanmoins généralement utilisés pour créer des *services* installés dans la couche système et qui donc devaient être toujours disponibles à l'ensemble des processus. Dans ce type de développement, il faut en effet éviter que le processus serveur se bloque sur une attente particulière, le processus doit pouvoir scruter les différentes requêtes.

Le noyau d'OMPHALE réalise uniquement le transfert asynchrone de messages de tailles variables. Il s'inspire fortement du schéma d'exécution des acteurs de CHORUS sans reprendre la notion de portes. Nous le décrivons en détails dans le chapitre suivant sur la description du noyau NERF d'OMPHALE. Nous voulons ici justifier ces choix.

1) La communication OMPHALE utilise le transfert asynchrone de messages.

Notre choix de la communication asynchrone est fondé i) sur la meilleure exploitation du parallélisme qu'elle permet (aspect création de systèmes) ii) sur son adéquation avec la modèle Serveur qui dit que toute entité a simplement pour rôle de traiter les messages reçus.

2) Les messages OMPHALE sont de tailles variables, et ne contienne, outre le

contenu du message, que les noms du module destinataire et du service demandé.

Cette simplicité est issue du modèle Serveur qui préconise qu'un serveur soit écrit indépendamment de qui l'utilise. Dans ce sens, l'émetteur du message n'a pas à s'identifier systématiquement dans le message. Les messages sont anonymes. Néanmoins si le nom de l'émetteur est nécessaire à la coopération, il peut être placée dans le contenu du message (on verra plus loin que les noms, qui apparaissent comme des capacités, peuvent être placés dans les messages).

3) Il n'y a pas dans la communication OMPHALE de support particulier pour le transfert de grandes quantités d'informations.

De tels transfert doivent être peu fréquents dans le modèle Serveur. En effet, Le modèle Serveur préconise l'encapsulation des données. Les messages véhiculent des demandes d'opérations sur les données et non pas en général les données elles-mêmes. Inversement, sur une base du modèle Producteur-Consommateur, les gros transferts sont fréquents.

4) Le transfert OMPHALE se fait de processus à processus sans intermédiaires de type portes.

Le stockage des messages en attente de traitement est réalisé essentiellement chez le processus destinataire dans une zone dite de réception. Cette zone est utilisée par le contrôleur du processus (voir le Modèle Serveur).

Il y a plusieurs justifications à la non utilisation d'intermédiaires de type portes :

– Comme on l'a déjà dit, la notion de porte montre tout son intérêt sur la base d'une coopération essentiellement de type producteur consommateur. Le modèle Serveur qui se fonde sur la coopération à base de la requête de service est donc d'une approche différente.

– Dans le modèle Serveur, le contrôleur du Serveur joue son rôle sur la base des messages arrivés au Serveur. Pour cela, le stockage des messages en attente est naturellement dans le serveur lui-même et non pas dans une entité autonome externe.

– Le fait de s'adresser à une porte sans connaître le ou les modules qui sont derrière la porte (ce qui fait l'intérêt majeur des portes comme intermédiaire) n'apparaît pas comme fondamental dans le modèle objet étendu au modèle Serveur. En effet, dans le modèle Serveur, on s'adresse à un serveur car on sait qu'il est capable de répondre à un certain type de requête à partir de certaines informations qu'il renferme. Dans la plupart des cas, un seul serveur remplit ces conditions voulues. La présence d'un intermédiaire parait alors superflue.

– Les aspects positifs des portes (possibilités de serveurs multiples ou anonymes, serveurs de remplacement en cas de pannes, etc...) peuvent être retrouvés par l'utilisation de modules serveurs intermédiaires qui réalisent en fait

les fonctionnalités des portes.

1.7.8 Le nommage et la protection

La problèmatique du nommage dans un système d'exploitation est celle de la désignation des entités apparaissant dans ce système. La problèmatique de la protection est d'éviter que n'importe quelle entité puisse désigner, et donc utiliser, modifier etc.., n'importe quelle autre.

1) Pour le nommage, les systèmes mettent en place un mécanisme de nommage (adressage) étendu à l'ensemble du système, par l'intermédiaire de "*noms*": un nom permet de localiser une entité de façon unique dans le système.

2) Pour la protection [LAM74], il s'agit d'éviter qu'une entité puisse utiliser illicitement des noms. Une solution est celle des "*noms locaux*" : Un processus utilise des noms qui n'ont de sens que pour lui pour accéder aux autres entités. Le système se charge de traduire, lorsque cela est utile, les noms locaux en noms globaux grâce à une table de traduction appartenant au contexte du processus.
Une entité ne peut donc utiliser que les entités pour lesquelles elle a un nom local qui peut être traduit en un nom global par le système. C'est une manière de limiter l'espace d'addressage des entités du système.

3) De manière plus fine, un nom local peut renfermer des "*droits*" qui indiquent le mode d'emploi autorisé de l'entité désignée. C'est la technique des Capacités [LEV84].

Nous détaillons ici ces trois points (voir aussi l'article de synthèse de [LEG88]) :

1.7.9 Noms globaux

Si au niveau de la machine physique on trouve des adresses mémoire (et au niveau réseau des adresses réseau), un système d'exploitation utilise toujours au dessus un système de désignation (par de "*noms*") qui lui est propre. Le but est de pouvoir désigner les entités indépendamment des spécificités du matériel (hétérogénéité) mais surtout indépendamment de leur localisation (en mémoire ou sur le réseau). Au dessous du système de nommage, sont réglés les problèmes de localisation et donc d'accès aux entités. Au dessus du système de nommage, la localisation et l'accès ne sont plus des concepts présents, seule la désignation reste.

Dans les systèmes répartis qui veulent créer une machine virtuelle unique à partir d'une architecture répartie, le nommage doit permettre la désignation uniforme d'une entité, qu'elle soit locale (c–a–d sur le même site que celle qui désigne), ou distante(c–a–d sur un autre site). On dit qu'il existe un nommage unique étendu à l'ensemble de l'architecture. On parle de **noms globaux**.

Le noyau d'un système d'exploitation réparti doit donc au moins créer ce

mécanisme de nommage étendu. En suivant [LEG88], un nom doit être **unique dans l'espace**: Un nom doit désigner une seule entité quelquesoit sa localisation (même en cas de migration), et un nom doit être **unique dans le temps** : Un nom désigne toujours la même entité ou si l'entité n'existe plus, le nom ne désigne plus rien. On parle alors de **noms uniques.**

1.7.9.1 Unicité dans l'espace

La propriété d'unicité dans l'espace, dans un système centralisé, est facilement obtenue par l'utilisation de tables assurant la traduction nom–adresse mémoire. Dans un système réparti, l'utilisation de tables globales n'est pas possible, et c'est donc en général grâce au nom lui–même que va se faire la localisation : Le nom comprend par exemple un champ précisant le site où se trouve l'entité désignée par ce nom. Ce problème est fortement compliqué si des possibilités de migration existe dans le système : Lorsqu'une entité migre (c–a–d change de site), les champs de site des noms dispersés pour désigner cette entité ne sont plus à jour. Différentes solutions permettent d'aborder ce problème (voir [POW83] pour un exemple). OMPHALE, dans sa version actuelle, ne s'est pas intéressé à la migration : un module serveur est crée sur un site et y reste. Des stratégies telles la diffusion ou des serveurs de localisation sont néanmoins des stratégies envisageables dans les versions ultérieures du système sans remettre en cause le mécanisme de désignation interne. Dans le noyau OMPHALE, un nom interne unique est fabriqué de manière décentralisée : le noyau du site où doit se faire la création fabrique un nouveau nom par concaténation d'un numéro de site et d'un numéro sur le site.

1.7.9.2 Unicité dans le temps

L'unicité dans le temps dit qu'un nom va désigner toujours la même entité, et que si cette entité disparaît, ce nom ne sera pas utilisé comme désignation d'un autre objet. On parle de noms **non réutilisables**. L'intérêt réside ici dans l'impossibilité d'erreurs dues à la réutilisation de noms par le système. On peut facilement créer des noms non réutilisables en utilisant des compteurs croissants pour fabriquer le numéro sur le site.

On peut aussi éviter ce genre d'erreurs par une solution duale: La destruction d'une entité ne peut être entreprise que lorsqu'il n'existe plus de noms dispersés dans le système pour cette entité. Dans ce cas, les noms peuvent être réutilisés sans craindre l'erreur précédente. La mise en place de cette solution nécessite des compteurs de références pour détecter les entités non référencés. En général, une technique de *'ramasse miettes"* est utilisée pour, donc, récupérer les noms réutilisables. Ces mécanismes sont difficiles à mettre en oeuvre en univers réparti (mis à jour de compteurs de références distants, ramasse–miettes réparti...), elle

pénalise globalement le système ou nécessite son arrêt pour faire régulièrement le nettoyage (un ramasse–miettes réparti à la recherche des boucles de références ne peut se faire actuellement en cours de fonctionnement du système). Une telle solution est de plus fortement sensible aux pannes (perte de compteur lors de panne de site, perte des messages de mise à jour des compteurs de références).

OMPHALE préconise l'emploi de noms globaux uniques non réutilisables, créés sur chaque site par la concaténation d'un numéro de site et d'un numéro sur le site obtenu par un compteur croissant. Ces noms globaux sont utilisés pour la désignation des modules serveurs, et cela uniquement dans le noyau NERF et dans le système de transport. La présence de ces noms globaux au niveau du système de transport entraîne un manque de sûreté du système complet. Des solutions utilisant des nombres choisis aléatoirement dans un espace important ("*large sparse space*") [TAN86] peuvent être trouvées dans les systèmes AMOEBA , V–KERNEL et CHORUS V3. La probabilité de créer de façon malveillante ou erronée un nom valide est ici virtuellement nulle. Une telle solution devrait être retenue dans les versions ultérieures d'OMPHALE.

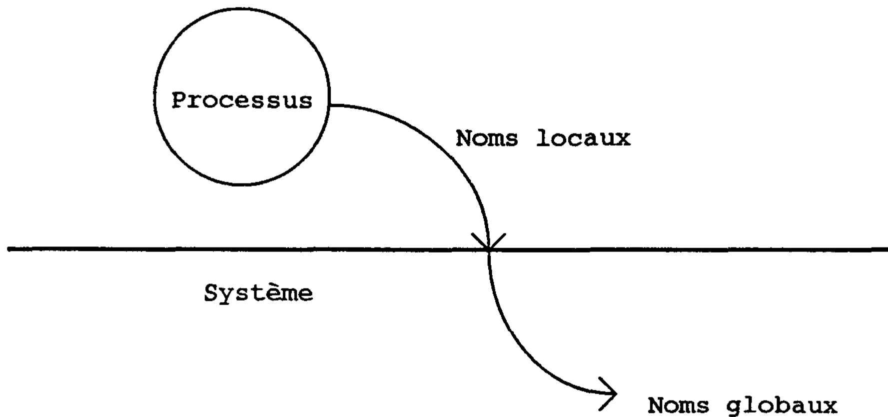
Un lieu de stockage des messages OMPHALE en attente existe dans chaque processus OMPHALE (c'est sa Zone de Réception). Contrairement à d'autres systèmes, le noyau d'OMPHALE n'utilise donc pas un autre espace de nommage pour des objets spécifiques de communication. Dans certains systèmes (CHORUS ou ACCENT), le lieu de stockage des messages est une porte, entité spécialisée du système. Un message est envoyé à une porte, un message est reçu d'une porte. Le noyau utilise donc pour les communications un deuxième espace de noms globaux pour les portes. Les noms globaux de processus ne sont alors jamais utilisés pour la communication.

On a déjà dit que cette notion de porte a la puissance du modèle producteur consommateur pour les processus communicants et qu'elle permet de jouer avec son aspect d'intermédiaire pour permettre toutes sortes de configuration ou reconfiguration derrière les portes. OMPHALE n'utilise pas ce concept, à la fois pour garder la simplicité d'un seul espace de noms globaux, mais aussi pour se placer au niveau du modèle Serveur différent du modèle producteur consommateur.

1.7.10 Noms locaux

L'utilisation de noms globaux permet la désignation des entités du système. Ils peuvent servir comme système de nommage, comme le fait V–KERNEL(le V–KERNEL utilise des noms globaux comme paramètres des ses primitives). Mais les noms globaux n'apporte pas de protection. Celle–ci doit être mise en place par un contrôle de la dissémination des noms globaux. Il reste alors le problème de la création directe de noms globaux par des processus malveillants ou erronés.

Une solution plus générale est l'utilisation de noms locaux: Un processus n'utilise que des noms qui lui sont propres, le système se chargeant de les traduire en noms globaux. On a donc un mécanisme d'indirection.



Noms locaux / Noms globaux

Chaque processus évolue donc dans un contexte qui lui est propre, implanté par une table de traduction Noms locaux —> Noms Globaux. Comme seuls les noms locaux sont utilisés au niveau processus et que de plus les noms locaux ne peuvent être obtenus que du système, ce mécanisme joue son rôle de protection : La visibilité d'un processus est limitée aux entités accessibles par les noms locaux valides qu'il possède.

Le prix à payer pour cette protection est dans la gestion des noms locaux par le système: la gestion des tables de traduction, la traduction elle-même et aussi le mécanisme nécessaire de traduction entre contextes des noms locaux lorsque ce ceux-ci participent aux paramètres des communications. Le problème général est ici l'**acquisition dynamique de noms locaux.**

Dans CHORUS V2, les acteurs désignent les portes (objets de communication du système) par des noms locaux dits "*contextuels*". Différentes opérations sont disponibles pour acquérir dynamiquement des noms contextuels:

1) Une opération *mssource* fournit un nom contextuel pour la porte source d'un message reçu.

2) Quant un acteur est créé, il hérite des noms contextuels de son père.

3) Les portes peuvent apparaître sous formes de noms symboliques associés à des fichiers spéciaux dans le système de gestion de fichiers. Une opération *writepgn* permet d'enregistrer un nom contextuel sous forme de nom symbolique; une opération *readpgn* permet d'obtenir un nom contextuel à partir d'un nom symbolique.

Dans CHORUS V3 , les portes sont nommées uniquement par les noms globaux (il n'y a plus de noms locaux gérés par le noyau). Ces noms globaux peuvent être transmis dans des messages. La protection de ces noms est réalisée par leur mode de génération aléatoire dans un champ de 128 bits.

Le noyau NERF d'OMPHALE associe à chaque module ce que l'on appelle un **Environnement** et qui est donc une table de traduction Noms locaux/Noms globaux. Dans le Modèle Serveur, les serveurs se désignent pour se faire des requêtes de service. L'environnement d'un module renferme donc les serveurs disponibles pour ce module. Ils sont utilisables par des noms locaux. Ici, les noms locaux peuvent être placés dans les messages. A ce moment le noyau assure une traduction des noms locaux pour l'Environnement du module destinataire : Le module destinataire reçoit un message contenant des noms locaux valides pour lui. On a ici de l'acquisition dynamique de noms locaux par message. Accent utilise le même mécanisme pour les noms locaux de ses portes. Le mécanisme de NERF est décrit dans le chapitre suivant.

1.7.11 Capacités

En fait, l'utilisation de noms locaux limite l'environnement d'une entité aux différentes entités que désignent ces noms locaux. En général, on va souhaiter avoir une protection plus fine en contrôlant aussi le mode d'emploi de l'entité désignée. On parle ici de **droits**. Le système doit gérer ces droits et en vérifier l'utilisation conforme à sa philosophie.

Dans le système Accent, le propriétaire d'une porte (celui qui en a demandé la création) a les droits de destruction, réception et émission pour cette porte. Ces droits peuvent là aussi être échangés dans les messages. Néanmoins le système assure qu'au plus un processus a , à un instant t, les droits de destruction et de réception sur une porte.

Classiquement une **Capacité** [FAB744] est une structure qui renferme un nom global (propriété d'adressage) et des droits (opérations permises) sur l'entité désignée (propriété de protection). Pour effectuer une opération sur un objet, il faut posséder une capacité pour cet objet qui autorise cette opération. Cela assure la protection voulue au niveau du mode d'emploi des entités du système. Les capacités elle-mêmes, doivent être protégées. Dans la plupart des systèmes, les capacités sont entièrement gérées par le noyau, ce qui assure leur protection. Le système AMOEBA au contraire, les crypte et donc peut les laisser dans l'espace utilisateur. Cela évite l'ensemble important des appels systèmes souvent nécessaires pour manipuler des capacités encapsulées dans le noyau.

L'emploi de capacités dans un univers d'objets est intéressante. En effet un objet est manipulable par l'intermédiaire d'une interface composée d'opérations. Dans les systèmes comme GUIDE, formés de processus et d'objets passifs, les capacités permettent de limiter sélectivement les processus dans la manipulation

des données. Dans les systèmes comme OMPHALE, formés d'objets actifs communicants, les capacités permettent de limiter sélectivement les modules dans l'envoi de requêtes de service. Les capacités jouent donc ici un rôle de **contrôle des communications**. Pour cette raison, nous appelons nos capacités "**Liens**" par référence au système DEMOS [BAS77] (puis DEMOS/MP [MIL87], Roscoe [SOL78] [SOL79] et PERSEUS) qui les introduisit pour le contrôle de la coopération de processus.

Un lien OMPHALE renferme 3 champs :

1) Un nom global de module

2) Un ensemble de service disponibles dans ce module

3) Un indicateur de droit de duplication du lien lui–même.

L'environnement d'un module renferme ses liens. C'est une structure encapsulée dans le noyau. Des primitives permettent, sous contrôle, de modifier les deux derniers champs.

La gestion des capacités a été abondamment étudiée dans la littérature [LEV84]. Les problèmes classiques sont ceux de la duplication, du transfert, de la révocation... Les solutions retenues dans le noyau NERF sont décrites dans le chapitre suivant.

1.8 Conclusion

Dans la première partie de ce chapitre, nous avons montré en quoi le Modèle Serveur est une manière d'introduire le parallélisme dans le Modèle Objet. Chaque objet (muni d'un contrôleur qui ordonne les messages reçus) devient une entité active (implanté par un processus) qui participe au système et applications

Le modèle proposé se veut simple et uniforme: Tous les processus ont un comportement de Serveur qui se caractérise par un traitement séquentiel des messages reçus.

La deuxième partie de ce chapitre montre que cette notion de Serveur s'inscrit très bien dans la problèmatique des systèmes répartis structurés en termes de processus communiquant par messages. Face aux choix de conception, généralement ouverts dans le panorama des systèmes existants, le Modèle Serveur apporte des éléments de réponse:

i) un objet par contexte d'exécution.

ii) la communication asynchrone (pas d'appel de procedure à distance mais simple transfert de messages).

iii) l'absence de portes (abandon du modèle producteur consommateur).

iv) pas de gestion des gros transferts (données encapsulées).

Les aspects nommage des systèmes répartis et en particulier assurer la transparence de la répartition permettent la distribution des serveurs sur un ensemble de sites. Cela rend possible une exploitation réelle du parallélisme de l'architecture.

La protection nécessaire dans un système se réalise bien dans le modèle Serveur par l'utilisation de Capacités qui contrôlent à la fois la désignation et l'utilisation des services des serveurs.

Le chapitre suivant décrit le noyau NERF issu des réflexions de ce chapitre. Il s'agit des spécifications générales d'un noyau de système réparti fondé sur le modèle Serveur. Trois remarques sont à faire:

1) Certains aspects des systèmes répartis tels la migration ou la tolérance aux pannes n'ont pas été abordés dans cette étude.

2) La validation de notre modèle ne pourra se faire que lors de la construction de la couche CORTEX du système. Ces travaux sont en cours actuellement sur la base de la plate-forme expérimentale Unix qui est décrite dans le chapitre 3. Ces travaux pourraient entraîner des modifications des spécifications du noyau, nous pensons qu'elles seront peu nombreuses et ne changeront profondément le modèle présenté ici.

3) On a dit que les performances à partir du Modèle Serveur vont être pénalisées par les fréquents changements de contexte issus de la coopération de nombreux processus. Un élément de réponse est la plate-forme expérimentale conçue spécifiquement pour NERF (le site 0) qui est décrite elle–aussi au chapitre 3.

Chapitre 2. Description du noyau NERF

Ce chapitre forme la description du noyau du système OMPHALE appelé **NERF**. NERF est un noyau destine a supporter le modele Serveur en univers reparti defini dans le chapitre precedent.. Les entités supportées par NERF sont des processus objets monoprogrammés (voir le chapitre précédent) que nous appelons Modules Serveurs ou plus simplement modules.

Ce chapitre décrit:

1) Le système de transport des messages OMPHALE qui se caractérise par son mode asynchrone et par son double espace de stockage des messages (avant et après émission c'est-à-dire à la fois chez l'émetteur et le récepteur) .

2) Le schéma d'exécution des modules Serveurs, qui correspond à celui de CHORUS V2, c'est à dire par enchaînement d'étapes de traitements autour d'une opération d'attente sélective de messages.

3) Le nommage et la protection engendrés par l'utilisation de Capacités ("Liens") entre modules.

4) Les mécanismes de création et destruction de modules.

Ce chapitre se termine par un aperçu de la couche CORTEX formée de Serveurs systèmes.

Il s'agit ici de travaux sur l'intégration des approches par objets au niveau du parallélisme et de la répartition. En ce sens différents aspects de la problèmatique des systèmes d'exploitation réparti ne sont pas étudiés ici, tels la tolérance aux pannes , la reconfiguration et la migration. Le NERF se veut être un **noyau simple et uniforme** et donc facile à mettre en oeuvre sur différentes architectures. Les couches supérieures du système (couche CORTEX) font partie des travaux actuels des membres de l'équipe OMPHALE.

OMPHALE est une architecture de système permettant:

1) De concevoir une application comme **un ensemble de processus coopérants par communications asynchrones par messages.**

2) De garder, au niveau système, le bénéfice des recherches sur la programmation modulaire, orientée objet et le génie logiciel. C'est-à-dire, essentiellement, de

concevoir une application comme un ensemble d'entités, les plus autonomes possibles, et chacune regroupant des fonctionnalités logiquement liées. Chaque entité est alors conceptuellement vue comme un **serveur.**

3) D'utiliser une architecture matérielle répartie comme support d'exécution.

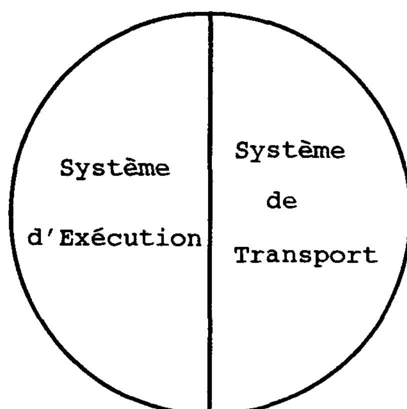
Pour bénéficier des acquis des nouvelles méthodes de décomposition par objets et du modèle Serveur, OMPHALE est un donc **système de processus.** Les processus OMPHALE s'appellent des **Modules Serveurs** et peuvent être perçus comme des "*objets actifs*" encapsulant des ressources propres (en particulier un espace mémoire privé physiquement inaccessible de l'extérieur), référençant d'autres modules, et offrant une "**interface**" sous forme de **services** à ceux qui désirent les utiliser.

Un module est l'unique entité reconnue par le noyau NERF. Un module est totalement résident d'un site de l'architecture, mais peut utiliser de façon transparente les modules de l'ensemble des sites. Ce noyau construit une machine virtuelle pour les couches supérieures (couche système et couche utilisateur) qui sont réparties sur l'ensemble des sites. NERF existe, quant à lui, identiquement sur chaque site.

Les modules OMPHALE sont des processus au sens classique du terme. Le premier rôle d'un noyau de système de processus est de faire la gestion des processus (allocation des processus au processeur, gestion des processus suspendus etc...). Un NERF prend donc en charge les modules locaux de son site. Plus particulièrement, c'est le rôle d'un sous-système de NERF appelé **Système d'Exécution** que nous décrivons dans ce chapitre.

Conformément au Modèle Serveur, chaque module ordonne les traitements à l'aide d'un **contrôleur**. Ce type de comportement peut facilement être réalisé à partir d'une opération **primitive d'attente sélective de message**. Un module effectue donc en alternance des **étapes de traitement** et des étapes d'attente de messages. On a donc un modèle d'exécution des modules qui s'apparente à celui de CHORUS V2 .

Les modules OMPHALE communiquent par envoi asynchrone de messages. Le NERF d'un site réalise la prise en charge des messages émis par un module local et le dépôt des messages reçus par un module local. Il agit donc comme un "*facteur*". Précisément, cette tâche est celle d'un sous-système de NERF appelé **Système de Transport.** Le transfert des messages de NERF à NERF est, quant à lui, réalisé par un sous-système spécialisé du CORTEX appelé **Station de Transport** qui encapsule le support de communication physique et en assure la fiabilité.



Le NERF d'OMPHALE

2.1 Le Système de Transport

La coopération entre modules OMPHALE se fonde sur le transfert asynchrone d'informations. La justification du mode asynchrone a été donnée au chapitre précédent.

L'information est transférée sous forme de **messages.** Un message contient l'indication du destinataire et les informations à transmettre. Ces informations sont de type "*valeur*", il ne s'agit jamais d' "*adresse mémoire*" puisqu'un module n'a accès qu'à sa mémoire privée. On verra dans le paragraphe sur le nommage que l'on peut aussi trouver des "*noms*" dans les messages, ces noms définissant l'espace inter–modules.

Un message ne comporte pas d'indication de l'expéditeur du message. Si cette information est nécessaire à la coopération des modules communiquants, c'est à la charge de ceux–ci de transmettre leurs identités dans le contenu des messages. Il en est de même pour toutes les informations qui pourraient servir à réaliser un type particulier de coopération. Notre souci est évidemment ici de sortir du noyau ce qui peut être réalisé à l'extérieur du noyau.

La taille des messages n'est pas ici fixée. Dans le modèle Serveur, les messages sont essentiellement des requêtes de service et ne servent pas à faire circuler des grandes quantités d'informations entre les processus. Au contraire, les informations sont toujours encapsulées dans des serveurs qui les gèrent. Ce taux bas de "gros" transferts est un avantage certain pour l'utilisation de messages de tailles variables. Pour la même raison, on ne trouve pas ici de mécanismes particuliers pour la réalisation de ces gros transferts de données.

Le **Système de Transport** achemine chaque message émis vers le module destinataire correspondant.

Dans le transfert complet d'un message d'un espace mémoire de module à un autre participent **cinq** entités : les deux modules communiquants, les deux NERFs des deux sites concernés (un seul NERF si la communication est intra-site) et la Station de Transport qui utilise le support de communication inter-site s'il y a lieu.

Un tel transfert se déroule en plusieurs temps:

1) **Préparation des messages à émettre** :
Chaque module possède un espace mémoire propre appelé **Zone d'Emission** qui est utilisée pour les envois de messages. C'est dans cette zone, à l'aide de procédures prédéfinies, que le module stocke les messages qu'il désire envoyer. Cette zone correspond à un **espace de stockage des messages avant transfert.**
Outre le fait que les Zones d'Emission définissent un espace de stockage (ce qui est nécessaire avec la communication asynchrone), l'intérêt de la Zone d'émission sera mis en évidence dans le chapitre sur le contrôle des communications où l'on verra que c'est dans cette Zone qu'est contrôlée la "*légalité*" des envois de messages. Ce contrôle s'effectue donc avant l'envoi réel, ce qui facilite grandement les procédures de reprise. Pour permettre ce contrôle, la Zone d'Emission d'un module est une structure interne au noyau accessible par le module par l'intermédiaire de procédures prédéfinies qui sont décrites plus loin.

2) **Emission et prise en charge par le Système de Transport** :
A son initiative, le module prévient le système que sa Zone d'Emission est prête. A ce moment le Système de Transport prend en charge les messages contenus dans cette zone. Ils ne sont alors plus accessibles par le module. Cette étape correspond classiquement à l'appel d'une primitive d'envoi.

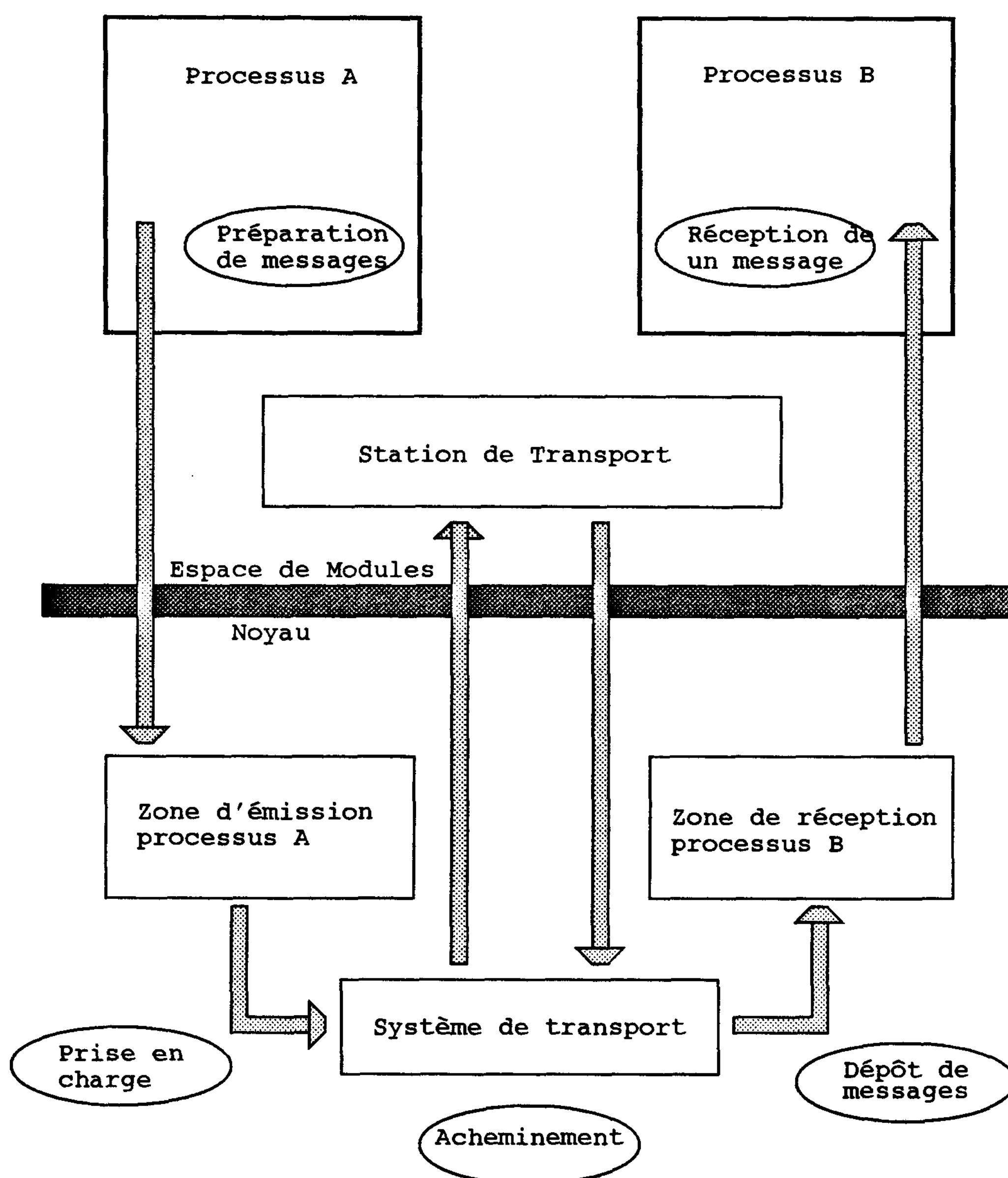
3) **Acheminement par le Système de Transport** :
Les messages pris en charge par NERF sont passés à un module du CORTEX appelé **Station de Transport** qui va déterminer le site du module destinataire et entreprendre le transfert physique du message vers le NERF du site concerné. Cette Station utilise le support de communication en utilisant un protocole adéquat pour obtenir la fiabilité voulue.
Le transport des messages est donc une tâche entreprise **à l'extérieur** du noyau.

4) **Dépôt des messages chez le destinataire** :
Chaque module possède une espace mémoire propre qui ne lui est pas directement accessible appelé **Zone de Réception.** C'est dans cette zone que le Système de Transport dépose les messages à destination du module. Cette zone correspond à un **espace de stockage des messages après transfert.** On verra que cette Zone est organisée comme un ensemble de files d'attente. L'intérêt est ici de permettre l'attente sélective de messages.

5) **Réception d'un message** : Lorsqu'un module est prêt à traiter un message, le

système lui fournit un message extrait de sa Zone de Réception. On appelle Réception le moment où le module obtient le message de la part du système, et non pas le moment où le message est déposé dans le Zone de Réception par le Système de Transport. Cette étape est centrale pour le schéma d'exécution défini dans ce travail et est décrite longuement dans le paragraphe suivant.

On peut résumer ces différentes étapes par le schéma suivant.



Le transfert de messages OMPHALE

Ce mode de transport des messages offre les avantages

1) de découpler fortement les modules communicants, ce qui est le but voulu pour la communication asynchrone.

2) d'offrir des interfaces clairement définies pour les modules vis-à-vis des communications: les communications se font ici par l'intermédiaire des Zones d'émission et de réception structurellement identiques pour tous les modules.

3) de reporter dans un module particulier (la station de Transport) la gestion des communications sur le support physique de communication, et donc en conséquence de s'adapter facilement à différentes stratégies d'utilisation du réseau pour obtenir une fiabilité et sûreté voulue.

2.2 Système d'exécution

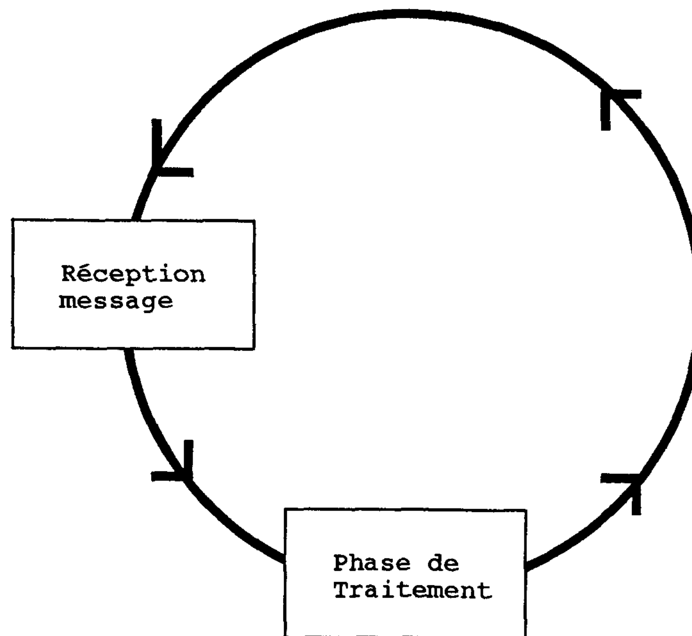
Le schéma d'exécution OMPHALE reprend celui de CHORUS V2 c'est à dire est construit à partir de la notion d'étape de traitement comme conséquence de la réception de message. Ce schéma simple est bien adapté au modèle Serveur car comme nous l'avons montré, il correspond à un fonctionnement de type "*automate*" dans lequel les messages traités entraînent des transitions dans un graphe des états possibles du module. Nous détaillons ici le cycle d'activité d'un module correspondant à ce schéma.

2.2.1 Réception et Traitement

Comme le montre la dernière étape du paragraphe précédent, un processus va traiter séquentiellement les messages qu'il reçoit. Le schéma d'exécution choisi pour un module est organisé autour de cette réception.

On définit une **phase de traitement** comme le travail effectué entre deux réceptions consécutives. Une phase de traitement est ininterruptible, elle se réalise de façon totalement indépendante des autres modules jusqu'à la prochaine demande de réception. L'arrivée d'un message à un module n'agit donc pas comme une "*interruption*", le message est simplement stocké en attente de traitement.

L'exécution d'un module doit donc être vue comme une simple séquence de phases de traitement.



Cycle Réception-Traitement

2.2.2 Emission

Une phase de traitement peut entraîner l'envoi d'un message de réponse vers l'émetteur de la requête. Une phase de traitement peut aussi entraîner l'envoi de un ou plusieurs messages de requêtes vers d'autres modules. Tous ces comportements résultent en l'envoi de messages comme conséquence du traitement.

Comme une phase de traitement est indivisible (en dehors des points de réception un module est un processus indépendant), les messages à émettre vont être retenus jusqu'à la fin de la phase de traitement en cours. Ils sont donc préparés dans sa Zone d'Emission par le module lui-même pendant la phase de traitement et envoyés à la fin de la phase de traitement.

Cela semble diminuer le parallélisme global de toute application. Néanmoins, plusieurs remarques viennent contre-balancer cette idée.

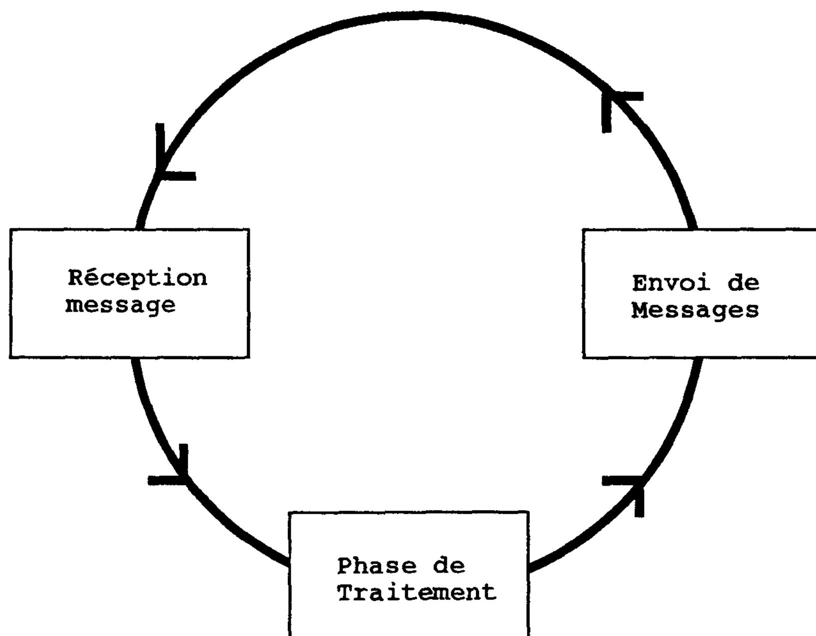
1) Une phase de traitement n'est pas en général une étape qui va être très complexe et très longue à s'exécuter. On peut le voir en regardant les programmes

modulaires type ADA ou des langages orientés objets. Les envois ne sont donc pas "*fortement*" retardés.

2) Si une phase de traitement entraîne l'émission de plusieurs messages, ceux–ci sont le plus souvent logiquement liés. Il n'est donc pas absurde de les envoyer simultanément (à la fin de la phase de traitement). De plus si un contrôle doit avoir lieu sur ces messages, il paraît intéressant de faire le contrôle globalement sur tous les messages et surtout d'éviter qu'un message ne soit envoyé alors qu'un message suivant, participant à la même activité logique, ne peut, lui, être envoyé. Une telle situation nécessiterait des procédures de reprise compliquées mettant en jeu tous les modules émetteurs et récepteurs. Ici ces contrôles vont se faire sur la Zone d'Emission et non pas lors de l'envoi.

3) Naturellement lorsqu'on envoie un message et que l'on attend la réponse, ce message doit être envoyé le plus tôt possible. Comme ici la réponse ne peut être obtenue que par la réception explicite d'un message, le mode envoi–réponse entre bien dans notre modèle dans lequel la fin d'une phase de traitement est justement le moment où l'on va trouver un enchaînement envoi–réponse.

L'exécution d'un module est donc une séquence de Réception–Traitement–Envoi.



Cycle Réception-Traitement-Envoi

2.2.3 Attente sélective

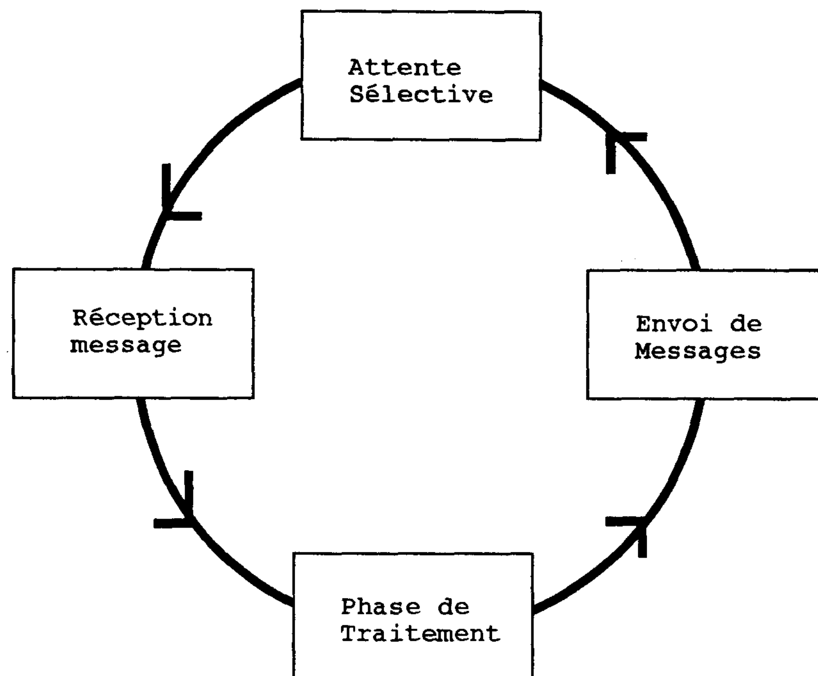
L'exécution d'un module apparaît donc dirigée par les messages qu'il reçoit. Elle est aussi dirigée par l'état interne du module.

Imaginons un module serveur de fonctions trigonométriques accessibles à tous. Chaque requête traitée par ce module comprend la fonction trigonométrique à calculer et les arguments nécessaires. Le traitement se termine par l'envoi d'un message de réponse. L'exécution globale de ce module est **entièrement** dirigée par les messages reçus. Il n'y a pas de corrélations entre les différents services car il n'y a pas d'état interne au module.

Reprenons le module serveur encapsulant un tampon. Ce module est accessible à des modules consommateurs qui émettent des requêtes de lecture et à des modules producteurs qui émettent des requêtes d'écriture. Normalement, le module serveur ne doit pas servir immédiatement les requêtes de lecture si le tampon est vide, ni les requêtes d'écriture si le tampon est plein. Il doit donc ordonner les traitements en fonction de son état (ici le remplissage du tampon).

Dans notre schéma d'exécution, cela signifie qu'un module doit pouvoir indiquer, au système et avant réception d'un message, quels sont les messages qu'il est prêt à traiter. Ce sont ses "*choix*" pour la prochaine phase de traitement.

Ainsi donc un module OMPHALE est, avant réception, dans un état d'**Attente sélective de message** d'où il ne sortira que lorsqu'il recevra un message correspondant aux choix signalés au système. Comme on le verra dans la section suivante, les choix sont signalés au Système d'Exécution qui a la charge de réveiller les modules en attente.



Cycle Réception-Traitement-Envoi-Attente

2.2.4 Système d'Exécution

Le Système d'Exécution de l'architecture OMPHALE organise le séquencement de l'exécution des modules.

Un module est dans l'un des deux états suivants :

1) **Actif** : Dans cet état, un module est virtuellement en train de traiter une requête (un "*allocateur*" de bas niveau se charge de faire partager le ou les processeurs entre les modules actifs). Un module actif ne peut subir de perturbations de l'extérieur. Il est totalement indépendant et ne communique ni se synchronise avec qui que ce soit.

2) **En Attente** : Dans cet état, un module est suspendu. Il est en attente de réception d'un message.

Le **passage de l'état Actif à l'état Attente** est réalisé par le module lui-même en exécutant une primitive de mise en attente. Cela correspond à la fin d'une étape de

traitement.

A ce moment les actions suivantes sont entreprises :

1) Mise à l'état Attente du module.

2) Prise en charge de la Zone d'Emission par le Système de Transport. Le Système de Transport récupère les messages préparés par le module pendant la phase de traitement qui se termine. La Zone d'Emission est vide pour la prochaine phase de traitement.

3) Prise en charge par le système des choix pour la prochaine phase de traitement. Il s'agit d'une "*liste de services validés*". Cette liste est stockée dans la Zone de Réception du module.

Le **passage de l'état Attente à l'état Actif** d'un module est à l'initiative du Système d'Exécution. Il peut être entrepris s'il existe dans la Zone de Réception du module, au moins une requête de l'un des services validés précédemment .

2.2.5 Description des Zones de Réception

On va ici décrire à l'aide de déclarations la Zone de Réception d'un module. Il s'agit de préciser sa structure permettant les mécanismes décrits plus haut. Ce ne sont pas des informations qui apparaîtraient au niveau d'une interface de programmation OMPHALE mais un guide d'implantation des données du noyau.

Introduisons d'abord la structure d'un message. Un message contient, outre le contenu du message, l'indication du destinataire et du service demandé. Ce couple (module destinataire, service demandé) est une "*adresse de service*".

```
TYPE
   adresse_service = RECORD
                       destinataire : ID_MODULE;
                       service : ID_SERVICE
                     END ;
   message = RECORD
               destination : adresse_service ;
               contenu : ANY
             END ;
```

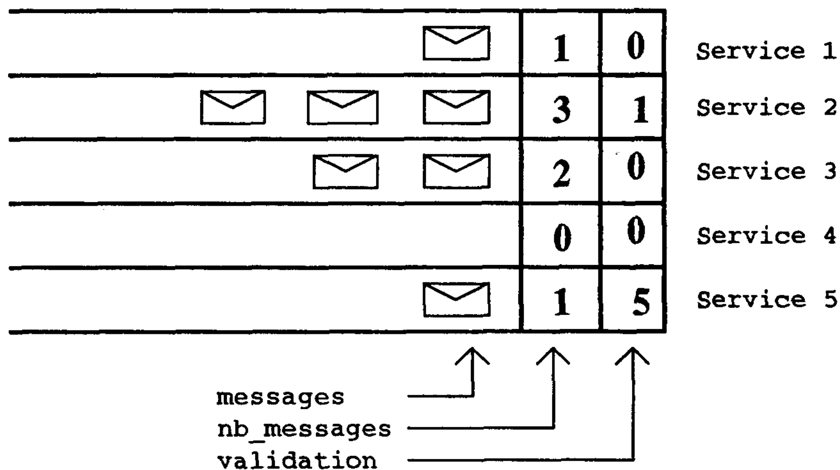
Les types prédéfinis ID_MODULE et ID_SERVICE qui apparaissent dans ces déclarations proviennent du mécanisme de nommage des modules et services dans le système OMPHALE. Ils seront détaillés dans le paragraphe étudiant le nommage. On supposera pour l'instant qu'il existe un mécanisme de désignation des modules et services étendu à l'ensemble du système.

Une des particularités du Système de Transport OMPHALE est l'interprétation par celui-ci du champ "*service*" des messages qu'il achemine. Ce champ va servir lors du dépôt des messages dans la Zone de Réception du module destinataire.

Une Zone de Réception contient autant de files d'attente de messages que de services que peut rendre le module. La Zone de Réception contient aussi les "choix" du module pour l'attente sélective.

D'une façon simple on peut décrire la Zone de Réception d'un module par les déclarations suivantes:

```
TYPE
   services = 1..MAX_SERV;
   priorite = INTEGER;
   queue = RECORD
            messages    : FIFO OF message;
            nb_message  : INTEGER ;
            validation  : priorite
           END ;
   zone_de_reception = ARRAY [services] OF queue ;
```



Zone de Réception d'un module

Le Système de Transport dépose un message dans la Zone de Réception du module désigné par le champ "*destinataire*" du message et en bout de la queue désignée par le champ "*service*" du message. Le "*nb_message*" pour cette queue est incrémenté.

Les "*validations*" sont mises à jour grâce à la "*liste de services validés*" fournie par

le module lors de sa mise en attente. Une validation non nulle correspond à un service validé pour la prochaine phase de traitement. Une validation positive est de plus interprétée comme une **priorité** associée à la file.

Une liste de services validés à la structure suivante :

```
TYPE
   services = 1..MAX_SERV;
   priorite = INTEGER;
   modetype = (null, force, augmente, diminue,....);
   liste_services_valides = ARRAY [service] OF
                                RECORD
                                  validation :  priorite;
                                  mode       :  modetype
                                END;
```

Une telle liste est passée au Système d'Exécution lors de la mise en attente du module. On va trouver différents modes de mise à jour des validations de la Zone de Réception en fonction de la liste fournie et du champ "*mode*". Quelques modes sont décrits ci dessous par la procédure de mise à jour des validations de la Zone de Réception.

```
FOR i IN services DO
  WITH la_zone_de_reception[i] DO
    CASE liste[i].mode OF
       null     : ;
       force    : validation := liste[i].validation;
       augmente: IF liste[i].validation > validation
                      THEN validation := liste[i].validation;
       diminue : IF liste[i].validation < validation
                      THEN validation := liste[i].validation;
       ...
    END;
```

Le Système d'Exécution fait passer un module à l'état Actif s'il existe une queue validée non vide, c'est-à-dire s'il existe q tel que :

```
( la_zone_de_reception [ q ] . validation <> 0 )
AND
( la_zone_de_reception [ q ] . nb_message <> 0 )
```

S'il existe plusieurs queues validées non vides, le système choisit la plus

prioritaire. En cas d'égalité de priorité, le système en choisit une de façon non déterministe.

Le système transfère ensuite le message en tête de cette queue dans l'espace mémoire du module pour la prochaine phase de traitement. Ce message est dit "*reçu*" par le module.

On peut résumer toutes les activités précédemment décrites par le schéma suivant:

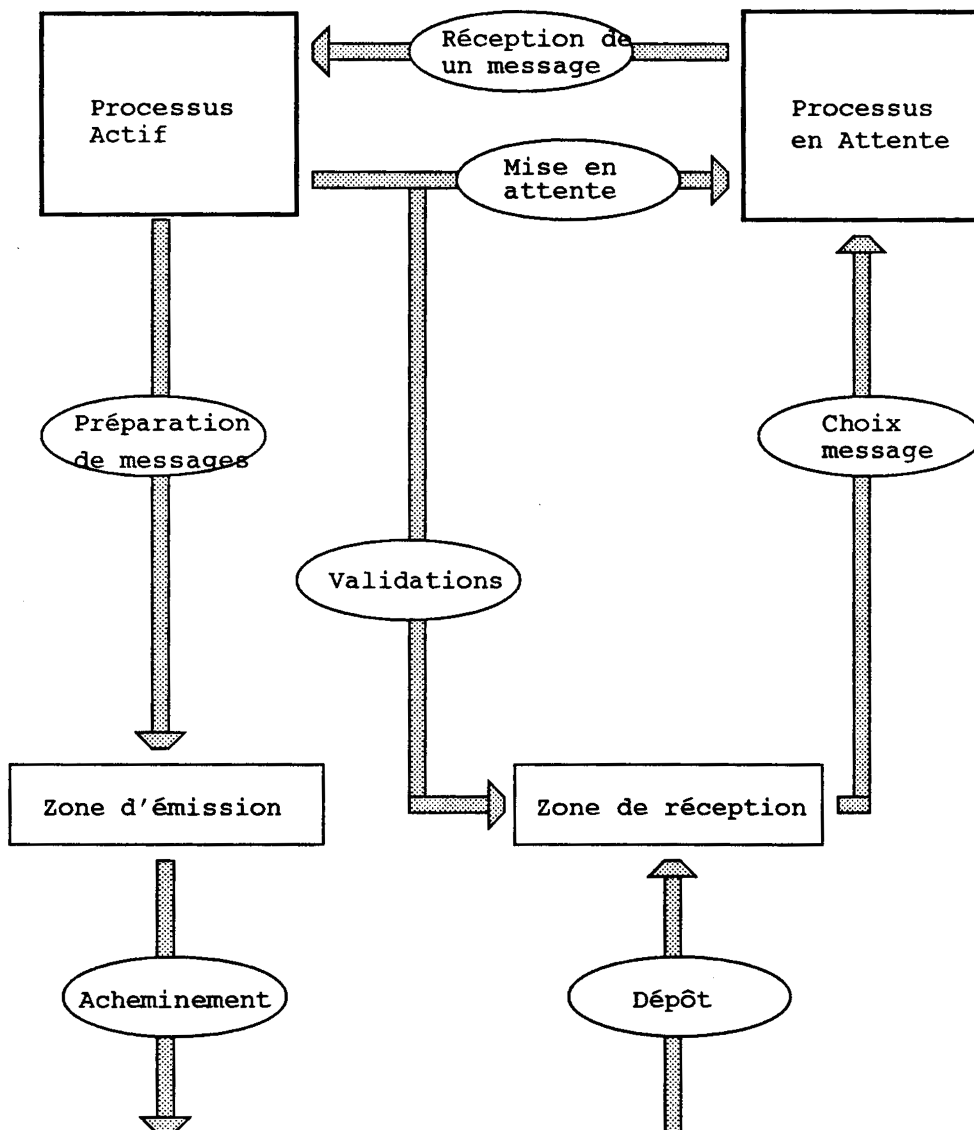


Schéma d'Exécution

2.2.6 Primitive unique d'envoi et de réception

L'envoi des messages préparés et le passage en attente avec indication des services validés se fait par l'appel d'une unique primitive que nous détaillons maintenant.

OMPHALE utilise une primitive unique d'envoi, d'attente et de réception de messages. Une seule primitive est nécessaire car ces trois actions sont toujours consécutives dans notre schéma d'exécution.

Cette primitive a la forme suivante, elle correspond à la fin d'une étape de traitement:

```
{ en-tête de la primitive ATTENDRE }
FUNCTION ATTENDRE(liste : liste_de_services_valides)
                 : message;
...
{ appel de la primitive }
message_recu := ATTENDRE(choix);
...
```

A l'appel de cette primitive, les actions suivantes sont entreprises:

1) Prise en charge des messages de la Zone d'Emission par le Système de Transport.

2) Positionnement des priorités de la Zone de Réception à partir de la "*liste_de_services_valides*" fournie.

3) Passage dans l'état Attente.

4) Attente du module jusqu'au réveil par le Système d'Exécution.

5) Récupération du message choisi par le système dans "*message_recu*".

6) Reprise de l'exécution derrière la primitive elle-même.

Cette primitive a pour résultat l'envoi des messages préparés pendant la dernière phase de traitement et l'attente et la réception du message à traiter pendant la prochaine phase de traitement.

Le module sait qu'il a reçu un message correspondant aux choix (multiples!) qu'il a fourni à la fin de la phase de traitement précédente. Le message reçu contient l'indication du service demandé. Et donc, si on fait suivre l'appel de la primitive d'une sélection de la procédure adaptée au service demandé, on obtient un

comportement de **Serveur** c'est à dire un objet muni d'un contrôleur, tel qu'on l'a décrit dans le chapitre 1.

L'exécution du service demandé s'apparente alors simplement à un appel de procédure interne au module.

```
message_recu := ATTENDRE(choix);
WITH message_recu DO
  CASE adresse_service.service OF
    s1 : service1(contenu);
    ....
    sN : serviceN(contenu);
  END;
```

Sélection d'un service

Ainsi donc un module serveur OMPHALE sera souvent construit sur le squelette suivant

```
MODULE squelette;
   VAR message_recu : message; s1, ... , sN : ID_SERVICE ;
   PROCEDURE service1(contenu:ANY);
      ...
   PROCEDURE service2(contenu:ANY);
      ...
   ...
   PROCEDURE serviceN(contenu:ANY);
      ...
BEGIN
   message_recu := ATTENDRE(choix);
   WITH message_recu DO
     CASE adresse_service.service OF
       s1 : service1(contenu);
       ....
       sN : serviceN(contenu);
     END;
END squelette;
```

2.2.7 Echéances

La primitive ATTENDRE peut être appelée en précisant une durée maximale

d'attente de réception de message. Cela évite l'arrêt définitif d'un module en l'absence d'arrivée de messages. Un tel mécanisme, dit d'"*échéance*", est indispensable dans le cadre d'une communication asynchrone, il est à la base des possibilités de détections de pannes (modules erronés, pertes de messages, arrêts de sites...). Il permet aussi d'aborder les problèmes de la programmation temps réel dans laquelle les entités doivent être très disponibles aux événements.

Le retour de la primitive ATTENDRE sur "échéance" se fait sur la "pseudo-réception" d'un message de requête d'un service prédéfini "**délai_expiré**. Cela s'exprime de la manière suivante :

```
...
...
message_recu := ATTENDRE_AVEC_DELAI(choix,10);
CASE message_recu.adresse_service.service OF
   s1 : service1(message.contenu);
   s2 : service2(message.contenu);
   delai_expire : { on n'a pas recu de message verifiant choix}
                  { pendant 10 ms,  on traite ce fait        }
                  ....
END;
...
```

Les délais sont classiquement pris en charge par NERF par la gestion de liste de processus en attente sur échéance.

2.2.8 Primitives de gestion de la Zone d'Emission

La Zone d'Emission d'un module est l'espace mémoire où il stocke les messages avant transfert. Le transfert a effectivement lieu à la fin de l'étape de traitement en cours.

La Zone d'Emission peut être simplement vue comme un espace pouvant contenir une liste de messages. Cette Zone est interne au noyau mais est gérée par le module lui-même au cours d'une phase de traitement. Cette gestion est réalisée grâce à un ensemble de primitives.

Cette technique rend possible le contrôle de l'utilisation de cette Zone et en particulier un contrôle de la *légalité* des messages placés dans cette Zone (voir le paragraphe sur le nommage dans ce chapitre).

Toutes ces fonctions manipulent des identifieurs de message de type prédéfini ID_MESSAGE (à la manière des descripteurs de fichier UNIX). Ces identifieurs peuvent se voir comme de simples indices dans une Zone d'Emission construite

autour d'une table de messages.

Ces fonctions contrôlent bien sûr la validité des ID_MESSAGE utilisés. On remarquera qu'un identifieur de message a une durée de vie limitée à une phase de traitement puisqu'à la fin d'une phase de traitement, la Zone d'Emission est vidée par le Système de Transport.

Ces fonctions contrôlent aussi le remplissage de la Zone. Celle–ci est de taille limitée, un module ne peut donc y stocker qu'un nombre limité de messages. Cela est une défense contre un module en boucle infinie sur le stockage de messages.

Un "*statut*" est retourné par ces fonctions, il signale les erreurs.

On trouve des fonctions telles les suivantes:

```
statut  := ECRIT_MESSAGE(mess, id_mess);
{ stocke le message "mess" dans la Zone d'Emission}
{ retourne un "id_mess" pour ce message }


statut  := LIT_MESSAGE(id_mess, mess);
{ extrait le "mess" de la Zone d'Emission }
{ il est désigné par "id_mess" }


statut  := DETRUIRE_MESSAGE(id_mess);
{ détruit le message désigné par "id_mess"}
{ de la Zone d'Emission }


statut  := DETRUIRE_TOUS_MESSAGES;
{ détruit tous les messages de la Zone d'Emission}
```

Fonction de gestion de la Zone d'Emission

On peut imaginer d'autres fonctions de manipulation de la Zone d'Emission (elle pourrait être organisée en pile de message par exemple), l'important est d'avoir la possibilité locale de revenir en arrière sur les messages déjà stockés.

Une telle possibilité peut paraître inhabituelle et inutile, cependant on s'aperçoit que souvent les messages préparés pendant une phase de traitement sont logiquement liés et que s'il est impossible d'en envoyer un il ne faut pas envoyer les autres. On peut parler ici d'*émision atomique* de tous les messages de la Zone d'Emission.

Un tel algorithme s'écrit ici :

```
.... calcul de mon_premier_message
ok := ECRIT_MESSAGE ( mon_premier_message, id1)
IF  ok = DESTINATAIRE_INCONNU THEN
       erreur(mon_premier_message,
              'destinataire inconnu')
ELSE BEGIN
       .... calcul de mon_deuxième_message
       ok := ECRIT_MESSAGE( mon_deuxième_message, id2)
       IF ok = DESTINATAIRE_INCONNU THEN BEGIN
             erreur(mon_deuxième_message,
                    'destinataire inconnu');
             st := DETRUIRE_MESSAGE(id1)
       END
     END
```

Exemple de reprise locale

On voit sur cette exemple que l'on met en place facilement une procédure de reprise qui est uniquement locale. Si le premier message avait été envoyé à tort, il aurait fallu mettre en oeuvre une procédure de reprise beaucoup plus compliquée, impliquant aussi le module ayant reçu le message envoyé.

2.3 Nommage

L'architecture OMPHALE définit les concepts de **module** et de **requête de service**. Le nommage définit ici les mécanismes permettant aux modules de se désigner pour se transmettre les requêtes de services. La notion de **protection** définit les mécanismes permettant au système de contrôler les interactions entre modules. Le nommage, s'il limite les possibilités de désignation, joue un rôle de protection.

Ce chapitre présente les mécanismes du noyau assurant le nommage et la protection. Ce sont des mécanismes de base qui permettent la réalisation d'un **système d'exploitation réparti protégé**.

Nous ne parlons pas ici des mécanismes de nommage et de protection de **fichiers**. Certains systèmes simples sont bâtis autour de la notion de fichiers, ce sont alors des "*Disk Operating Systems*" et les mécanismes de nommage et de protection apparaissent alors dans la désignation des fichiers. OMPHALE est bâti autour de la notion de modules (processus) et les mécanismes de nommage et protection apparaissent dans le désignation des modules. S'il doit exister, dans OMPHALE, des mécanismes de contrôle des fichiers, cela se fera uniquement dans le **Système de Fichiers** du CORTEX et non pas dans le noyau NERF.

Le niveau inter-modules OMPHALE est utilisable par la requête d'un service d'un

module. Le noyau définit des mécanismes de nommage des modules et des services:

2.3.1 Nommage des modules

Lors de l'exécution d'une application sur OMPHALE, deux systèmes de nommage sont utilisés: un mécanisme interne au noyau dans lequel chaque module possède un **nom unique**, et un mécanisme local à chaque module dans lequel les modules connus sont désignés par des **noms locaux**.

2.3.1.1 Noms uniques

Le système a la charge de l'ensemble des modules . Chaque module est désigné par un **nom unique** non réutilisable. Ce nom unique joue le rôle d'adresse globale sur l'ensemble du système. Il permet d'identifier le module et permet aussi de le localiser sur le réseau. Pour cela il comporte un champ **Numéro de site**. Grâce à ces numéros de site , ces noms peuvent être créés localement sur chaque site sans avoir besoin d'un mécanisme centralisé de création de noms uniques.

Numéro de site	Numéro local au site

Nom unique de module

Ce nom unique est essentiellement utilisé par le Système de Transport pour réaliser le transfert des messages. Un module n'utilise jamais les noms uniques.

2.3.1.2 Noms locaux

Pour des raisons évidentes de protection, il faut que chaque module ne connaissent qu'un nombre limité de modules: Ceux qui lui sont nécessaires pour effectuer son travail. Dans NERF, les noms uniques des modules connus d'un module sont regroupés dans une structure qui lui est associé appelée l'**Environnement** du module. Pratiquement, un Environnement est une **table de noms uniques**. Ainsi donc il n'existe pas de "*table globale*" des noms uniques, mais une table de noms uniques est associée à chaque module. Si deux modules A et B connaissent le module C, les environnements de A et B contiennent le nom unique de C.

Pour envoyer un message, un module doit désigner un module destinataire et donc

donner un nom unique. En fait il va simplement donner un **nom local** de module qui est un indice dans sa table des noms uniques. On parle de nom local car un tel nom n'a de sens que pour lui.

Les conséquences de ce mécanisme sont:

1) Un module ne connaît pas l'existence du niveau de nommage par noms uniques, ce qui est souhaitable car ce niveau est interne au noyau.

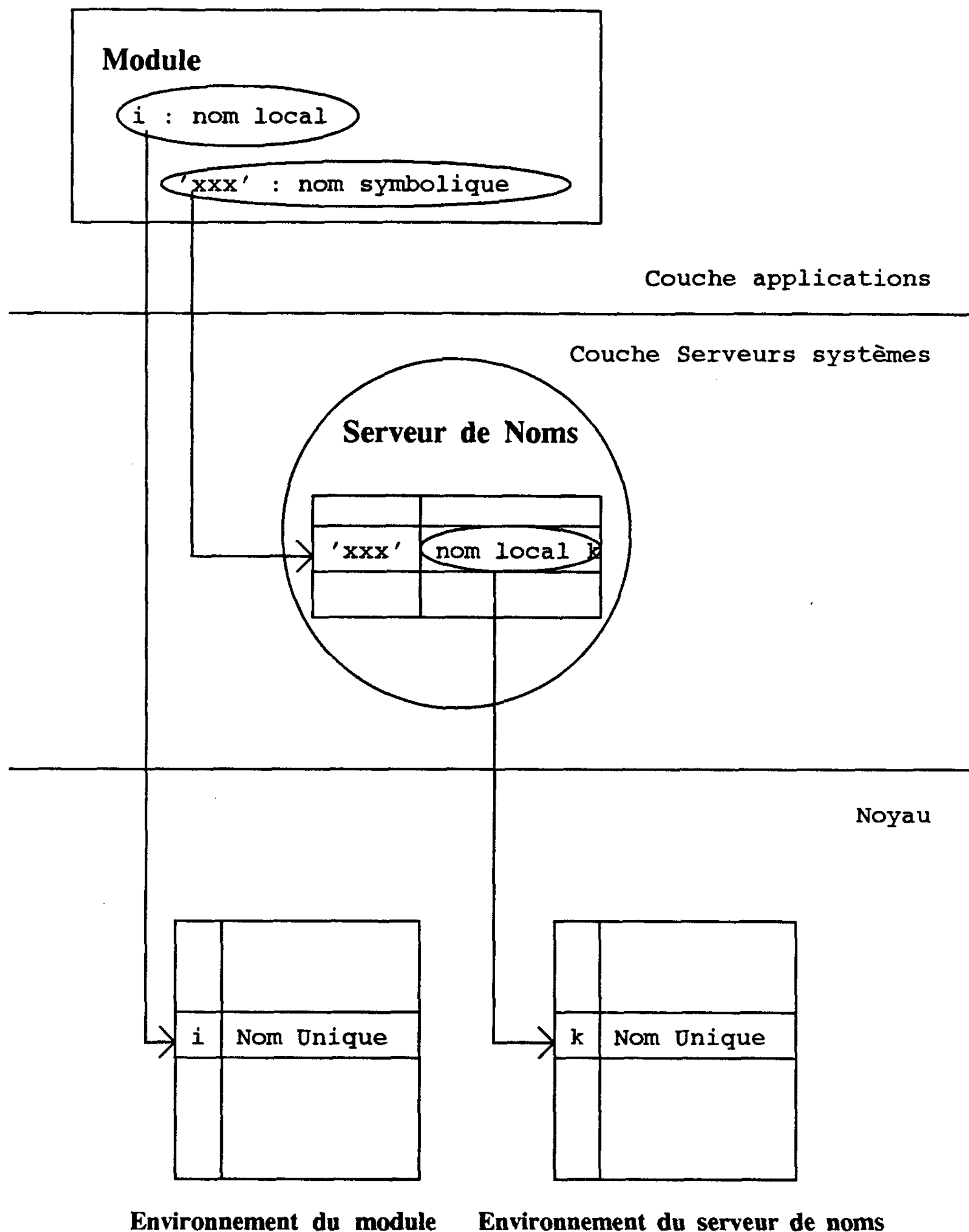
2) Un module ne se préoccupe pas de la localisation des autres modules sur l'ensemble des sites, cela assure la transparence de la répartition pour l'envoi de messages.

3) Un module a une vision limitée à son environnement du reste du système, c'est donc une implantation du Principe du Moindre Privilège [LIN76]. C'est aussi un "*environnement d'exécution fermé*" au sens de [DEN76]: ce qui est autorisé est explicitement précisé comme tel.

2.3.1.3 Noms symboliques

Un troisième système de nommage existe sur OMPHALE, c'est le nommage symbolique. Ce système de nommage n'appartient pas au noyau, mais est complètement pris en charge par des **serveurs de noms** du CORTEX.

Lors de la construction d'une application OMPHALE, l'utilisateur peut être amené à utiliser des modules existants qu'il ne connaît que sous un nom symbolique. Au cours de la compilation, il n'est pas toujours possible (et pas souvent souhaitable) de transformer ces noms symboliques en noms uniques. La solution retenue est alors de faire de la **liaison dynamique** pendant l'exécution entre noms symboliques et noms uniques. Cette liaison dynamique est assurée dans OMPHALE par des modules spécialisés du CORTEX appelés **Serveurs de Noms**. Un tel module est capable, pour un module, de fournir en retour un nom local à partir d'une requête précisant un nom symbolique. A la fin de ce chapitre, on reparlera de ces serveurs du CORTEX.



Différents niveaux de nommage

2.3.2 Nommage des services

Le Système de Transport reconnaît trois champs dans un message. Outre le contenu du message, on trouve :

1) Le **Module destinataire** : Ce champ identifie le module devant recevoir le

message et permet l'acheminement du message. Le module émetteur utilise des noms locaux, le Système de Transport utilise des noms uniques obtenus dans l'Environnement du module lors de la prise en charge des messages.

2) Le **Service demandé** : Ce champ identifie la requête. Il permet au Système de Transport de déterminer la file où stocker le message dans la Zone de Réception du module destinataire. Un nom de service est simplement un indice dans la table de queues de la Zone de Réception du module destinataire.

La communication inter-modules utilise donc un couple d'indiees (m,s) où m est un indice dans l'environnement local pour désigner le module destinataire et s est un indice dans la zone de réception du module destinataire pour désigner une file de requête. Un tel couple est appelé **Adresse de service.**

2.4 Protection

Les mécanismes de protection [LAM74] doivent assurer l'étanchéité d'une application. En respectant le **principe du moindre privilège**, un module ne doit avoir accès, à un instant donné, qu'aux services qui lui sont nécessaires pour continuer sa tâche.

Les **droits** d'un module sont définis par les services accessibles, à un instant donné, par ce module. La requête d'un service se formulant uniquement dans la communication entre modules, le contrôle des droits prend naturellement place dans le contrôle des communications inter-modules.

Les mécanismes de protection introduits au niveau du noyau OMPHALE vont permettre le contrôle local (c'est-à-dire au niveau de l'émetteur) de la légalité de l'envoi d'un message.

Plus précisément, il s'agit de vérifier la légalité d'une adresse de service et donc de répondre aux questions:

1) Le module émetteur a-t-il le "*droit*" d'envoyer un message au module destinataire?

2) Le module émetteur a-t-il le "*droit*" d'utiliser ce service du module destinataire?

La technique utilisée dans OMPHALE pour pouvoir répondre à ces questions est celle des "**Capacités**" [LEV84] que nous appelons des "**Liens**" car il s'agit ici de capacités entre processus telles que les introduisit le système DEMOS. Toute la problématique des capacités (modifications, transferts, révocations) se retrouvent dans les liens. Les paragraphes suivants décrivent les solutions retenues dans le noyau NERF.

Le contrôle de validité du contenu des messages n'est pas pris en charge par les mécanismes décrits ici. Il s'agirait pour cela de contrôler la conformité du contenu

du message avec une spécification de la communication (nombre et types des éléments du messages par exemple). Cela relève de techniques de plus haut niveau (compilation...) ou cela doit être à la charge explicite du programmeur des modules coopérants. Néanmoins, comme on le verra plus loin, les messages OMPHALE sont "*structurés*", et cela est une base importante pour un éventuel contrôle de haut niveau du contenu des messages.

2.4.1 Liens

Le mécanisme de communication inter-modules utilise la notion de **Liens** . Cette technique a été introduite dans le système DEMOS [BAS77] comme alternative aux sémaphores pour des machines dépourvues de mécanismes performants de protection mémoire. Elle a été reprise avec bonheur pour des architectures de machines sur réseau [MIL87] [SOL78] . La notion de Liens est issue de travaux antérieurs sur le concept de **Domaine de Protection** [DEN66] et sur les **Systèmes à Capacités** [LEV84].

Les Liens OMPHALE permettent le contrôle des adresses de service.

1) Un **Lien** définit une route unidirectionnelle entre deux modules. Un module A ne peut envoyer un message à un module B que s'il existe une route directe de A vers B, c'est-à-dire que si A possède un lien vers B. Pratiquement, un lien encapsule un nom unique de module.

2) Un **Lien** définit l'ensemble des services disponibles dans le module destinataire. Un module A ne peut envoyer une requête du service S dans le module B que si le lien vers B qu'il possède indique que le service S est disponible. Pratiquement un lien contient la liste des noms des services disponibles chez le module associé au lien.

L'ensemble des liens que possède un module définit donc l'ensemble de ses **droits**, c'est-à-dire l'ensemble des adresses de services qu'il peut utiliser pour envoyer des requêtes. Cela correspond à l'**Environnement** déjà défini qui apparaît donc maintenant comme une **Table de liens.**

Pour des raisons évidentes de protection, les liens que possède un module, ne sont pas directement modifiables par le module lui-même. S'il pouvait directement modifier le contenu de son environnement, un module pourrait franchir toutes les barrières de protection. L'environnement d'un module est ici pour cette raison une structure interne au noyau et un module va être limité à un ensemble d'opérations prédéfinies du noyau comme seule interface avec son environnement. Une autre solution peut être trouvée dans le système AMOEBA par un cryptage des capacités. Celle-ci peuvent alors être manipulées dans l'espace utilisateur.

Les **noms locaux** vus précédemment référencent donc ici les liens de l'environnement, c'est-à-dire de la Table de Liens.

Lorsqu'un module envoie un message, il doit donc préciser

1) **Le nom local du lien utilisé** : Le système peut tester si le nom local désigne bien une entrée valide de l'environnement.

2) **Le service demandé** : Le système peut tester si le service est bien indiqué comme disponible dans le lien considéré.

Si tous ces tests sont passés avec succès, le message peut être stocké dans la Zone d'Emission. Dans la Zone d'Emission, le champ "*destinataire*" d'un message est toujours le nom local utilisé par le module; ce ne sera qu'au moment de la prise en charge de la Zone par le Système de Transport que celui–ci transforme à l'aide de l'environnement les noms locaux de destinataire en noms uniques de modules.

Le choix de la notion de Liens est basé sur plusieurs constatations:

1) Dans le mécanisme de communication OMPHALE, les messages sont d'abord préparés dans la Zone d'Emission du module avant d'être réellement envoyés à la fin de la phase de traitement en cours. L'utilisation de Liens permet un contrôle des communications uniquement du côté de l'émetteur et de plus, la validité (au sens précédent) d'un envoi de message peut être contrôlée au moment du stockage du message dans la Zone d'Emission, c'est à dire **avant** l'envoi réel. Cela facilite beaucoup d'éventuelles procédures de reprise.

2) Les travaux sur les systèmes à capacités ont montré l'intérêt de tels systèmes pour la gestion de ressources [KIE78]. Le modèle Serveur généralise cette notion. Les Liens, qui sont des capacités entre processus, permettent de bénéficier de ces avantages.

3) On a vu que dans une architecture répartie, un mécanisme de communication asynchrone permet d'obtenir une plus grande indépendance des entités coopérantes. Le concept de Liens qui définit des routes unidirectionnelles entre modules est bien adapté à cette communication asynchrone.

4) L'utilisation de liens par l'intermédiaire de noms locaux donne la propriéte de transparence de la répartition à l'architecture. La manipulation des noms uniques est uniquement à la charge du Système de Transport.

2.4.2 Transfert de Liens

Le mécanisme de communication doit permettre tout type de coopération entre modules. Cela nécessite des possibilités de **transfert de liens** dans les messages.

Quelques exemples simples montrent la nécessité d'un transfert de liens :

1) Si un module A possède un lien vers un module X et désire donner dynamiquement la possibilité à un module B d'utiliser des services de X (c'est le cas lorsque A est un gérant du module X), A doit alors passer à B les adresses

correspondantes de service et cela ne peut se faire que par un transfert d'un lien vers X de A à B.
On remarque que pour que A ne perde pas la possibilité d'utiliser X, il faut lui donner la possibilité de dupliquer son lien vers X avant d'envoyer à C un des duplicatas.

2) Si deux modules A et B désirent dialoguer, ils doivent posséder chacun un lien vers l'autre. Ces liens ne peuvent être obtenus que dynamiquement par ces modules car au moment de la création de A (resp. B) le module B (resp. A) peut ne pas exister et le mécanisme de démarrage du module A (resp. B) ne peut créer le lien voulu. D'une manière générale, dans une telle situation les liens vont être obtenus à partir des **serveurs de noms** grâce à des transferts de liens des serveurs vers les modules A et B.

Dans OMPHALE, on ne va pas trouver des instructions spécifiques de transfert de lien, mais un mécanisme de "*traduction*" des noms locaux lorsqu'un module place de tels noms dans les messages qu'il envoie. Le transfert des liens correspondants a bien lieu, mais est complètement caché derrière l'utilisation des noms locaux.

Examinons attentivement les mécanismes mis en jeu pour les liens dans toutes les phases du transfert de messages.

1) **Stockage d'un message dans la Zone d'Emission**:
Un module peut utiliser les noms locaux comme des valeurs qui peuvent faire partie du contenu des messages.
Lors du stockage d'un message dans le Zone d'Emission, le noyau recherche les noms locaux qui pourraient se trouver dans le contenu du message (voir plus loin). Lorsqu'un nom local est ainsi trouvé, le noyau teste la validité du lien correspondant. En cas de succès, le noyau remplace le nom local par le lien correspondant. En réalité, le lien n'est pas mis à ce moment là dans le message (de la même manière que le destinataire n'est pas immédiatement remplacé par le nom unique), mais l'entrée correspondante de l'environnement est simplement "*marquée*" comme contenant un lien à placer dans un message. Cette méthode facilite les procédures de récupération de messages (du type `detruire_message`). Néanmoins, pour le module, le lien est dorénavant invalide (c'est-à-dire inutilisable par la suite) et pour lui cela s'apparente à la disparition du lien de son environnement et à sa présence dans le message.

2) **Destruction de message de la Zone d'Emission**:
La destruction (par `detruire_message`) d'un message de la Zone d'Emission rend de nouveau valides les liens qui avaient été placés dans le message.

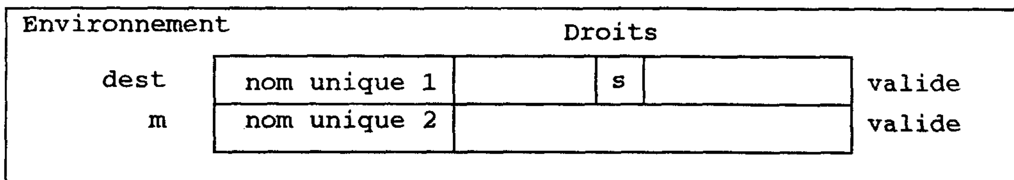
3) **Transfert par le Système de Transport**:
Lorsque le Système de Transport prend en charge les messages, il remplace les noms locaux présents dans le message (voir plus loin) par les liens de l'environnement. Ces liens **disparaissent** donc de l'Environnement de l'émetteur. Les messages sont déposés ainsi dans les Zones de Réception des modules

destinataires.

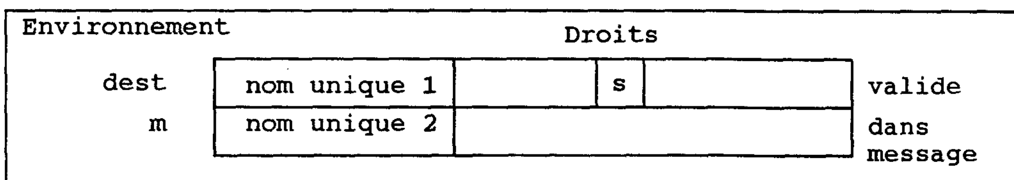
4) **Réception de message**:
Lorsque le Système d'Exécution fournit un message à un module les liens y apparaissant sont remplacés par des noms locaux correspondant à des places `vides` de l'environnement. Les liens **apparaissent** donc dans l'Environnement du receveur. L'entrée correspondante passe à `valide`. Pour chaque lien arrivant apparait donc un nouveau nom local utilisable par le module.

Illustrons ces mécanismes:

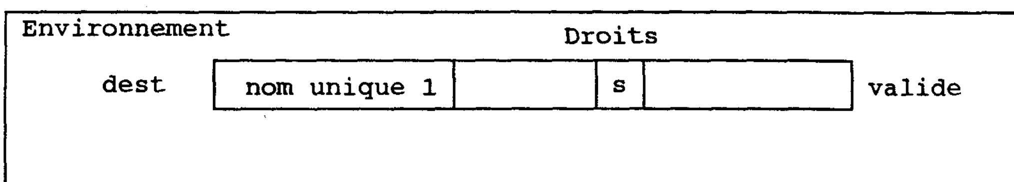
```
MODULE expediteur ...;
  ...
  VAR dest(s:ID_SERVICE), m:ID_MODULE;
  ...
  BEGIN
```




```
    ...
    ecrit_message(dest,s,m);
```

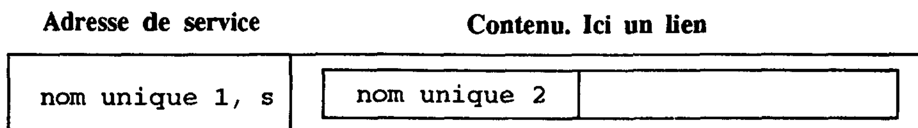



```
    ...
    ...ATTENDRE... {fin de l'etape de traitement}
```

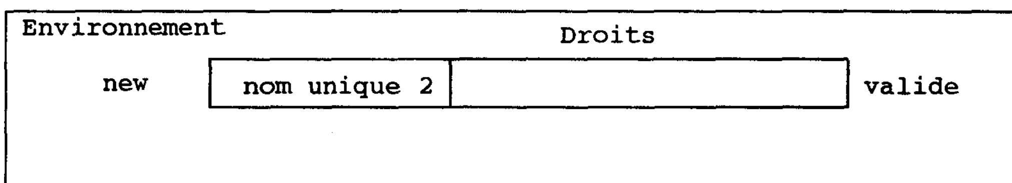



```
  END
```

Le message transféré par le système de transport est:




```
MODULE destinataire...;
   ...
   VAR new:ID_MODULE;
   ...
   BEGIN
      ...
      ...ATTENDRE... { Réception du message }
      new:=reception.contenu
```




```
    END
```

La technique précédente impose que le noyau puisse retrouver dans un message utilisateur les noms locaux y apparaissant pour les transformer en liens (à l'émission) et inversement, dans un message interne, les liens pour les transformer en noms locaux (à la réception). Cela signifie que les messages OMPHALE sont **"structurés"**. Cette structuration fait partie des spécifications du noyau OMPHALE et doit donc permettre ce qui vient d'être décrit.

Dans la version SPS7 (voir le chapitre 3), les messages sont composés dans l'ordre:

1) De la taille du texte utile

2) Du texte utile, c'est-à-dire du contenu du message hors liens,

3) Du nombre N de liens présents dans le message

4) Des N liens (qui sont de taille fixe).

Tout noyau OMPHALE doit utiliser une structuration des messages. Cette structuration doit aussi être choisie en vue de la construction de langages de haut niveau directement utilisables pour la programmation fiable des modules (on parle des problèmes d'"emballage" et de "déballage" ("*stubs*") des données pour le transfert) (voir aussi XDR de Sun [SUN86a]). A ce propos Jean-Marie PLACE de l'équipe OMPHALE travaille actuellement à un préprocesseur Modula-2 [PLA88] pour la version OMPHALE du site expérimental "Site 0" dans laquelle chaque message est formé du contenu du message précédé des positions dans le message

où se trouvent les liens.

Tous les mécanismes précédents permettent le transfert de liens d'un environnement d'un module vers un autre environnement. Cette possibilité, pour être pratiquement utilisable, doit s'accompagner d'autres possibilités telles la duplication de liens (déjà introduite), le contrôle de l'utilisation des services et la destruction de liens. Nous les introduisons maintenant.

2.4.3 Duplication de Liens

Un module peut **dupliquer** un lien dans son environnement. Celui–ci contient alors *deux* liens identiques vers le même module. Le duplicata obtenu est utilisable par un nouveau nom local. Cette opération doit permettre au module A de conserver un lien vers un module X après transfert d'un tel lien au module B.

La duplication est toutefois limitée par un *indicateur* présent dans chaque lien qui autorise ou interdit les duplications du lien. Ce contrôle permet à un module de passer un lien à un autre module en étant assuré que ce dernier ne dupliquera pas le lien. Cela est en général très utile pour un gérant de ressources qui ne souhaite pas une dispersion incontrôlable des liens vers les ressources qu'il gère.

En conséquence, un module peut rendre *non duplicable* un lien duplicable de son environnement, mais l'inverse n'est pas possible. Notons aussi que la duplication d'un lien crée un lien duplicable.

Les opérations autour de la duplication de liens sont les suivantes. Il s'agit de primitives du noyau OMPHALE:

```
FUNCTION DUPLIQUER ( m1,m2 : ID_MODULE) : ID_ERREUR;

FUNCTION NON_DUPLICABLE ( m : ID_MODULE) : ID_ERREUR;

...

status := DUPLIQUER (m1,m2) ;
{ m2 devient le nom local du lien dupliqué à partir}
{ du lien nommé par m1.}
{ status signale une erreur si m1 n'est pas duplicable}

...

status := NON_DUPLICABLE (m2) ;
{ m2 devient non duplicable }
{ status indique une erreur si m2 déjà non duplicable}
```

Primitives dupliquer et non_duplicable

2.4.4 Contrôle de l'utilisation des services

Un module doit pouvoir, sous contrôle, manipuler les listes de services indiquées dans les liens qu'il possède.

Par exemple, lorsque un module passe un lien vers lui-même pour pouvoir recevoir une réponse, il est naturel de ne valider dans ce lien que le service de réponse. Ainsi donc, le module est assuré que le lien ne sera pas utilisé d'une manière non prévue.

Par ce contrôle, un gérant de ressources peut passer à ses clients des liens offrant des possibilités différentes d'utilisation des ressources. Il définit ainsi des classes d'utilisateurs des ressources.

De plus, un module peut souhaiter que le lien qu'il passe ne soit utilisé qu'une fois par le receveur, c'est le cas lors qu'on passe un lien ne devant servir qu'à envoyer un message de réponse.

A partir d'un lien, un service peut être:

1) Inaccessible . C'est donc une erreur que d'essayer d'envoyer un message en utilisant ce lien et ce service.

2) Accessible une fois. C'est-à-dire que le module possédant ce lien ne peut envoyer qu'un seul message de requête de ce service.

3)Accessible librement . C'est-à-dire que le module possédant ce lien peut envoyer toutes les requêtes de ce service qu'il désire.

Pour un envoi d'un message de requête d'un service, si ce service n'est utilisable qu'une fois, son état passe à `déjà_utilise` dès le stockage du message dans la Zone d'Emission. Il n'est donc plus utilisable. Néanmoins si le message est détruit de la Zone d'Emission, le service reprend son état initial, c'est-à-dire `utilisable_une_fois`. A la fin de l'étape de traitement en cours, tous les liens dans l'état `déjà_utilisé` passent naturellement dans l'état `inutilisable`.

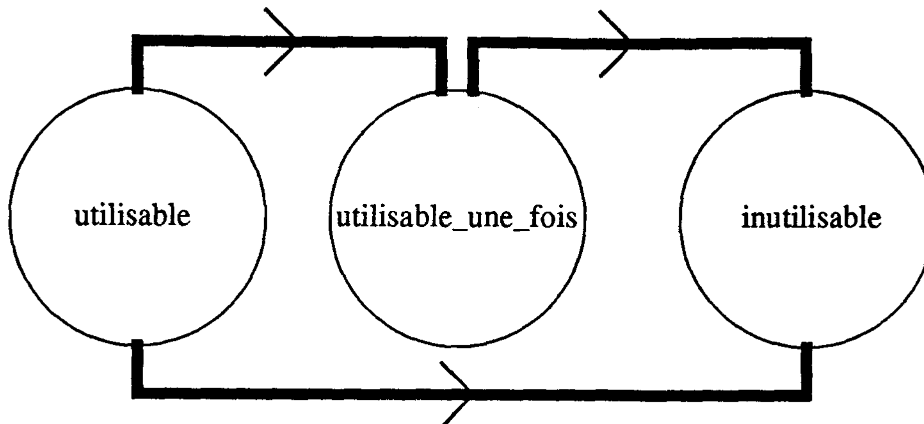
Des opérations vont permettre de restreindre l'utilisation des services des liens que possède un module. En aucun cas un module ne peut accroître l'utilisabilité des services qu'il référence.

Les seules transitions possibles d'état d'un service sont donc:

1) De `utilisable` à `utilisable_une_fois`

2) De `utilisable` à `inutilisable`

3) De `utilisable_une_fois` à `inutilisable`



Cela est réalisable par l'utilisation de la primitive suivante:

```
FUNCTION MODIFIER_ETAT_SERVICE( m : ID_MODULE,
                                s : IS_SERVICE,
                                e : etat_service )
                              : ID_ERREUR;
....

status:=MODIFIER_ETAT_SERVICE(m,s,e) ;
{ L'état du service s de m passe a e }
{ status signale une erreur si la transition}
{ à effectuer est impossible }
```

Primitive modifier_etat_service

La duplication d'un lien ne doit pas permettre d'acquérir de nouveaux droits. La duplication d'un service utilisable une fois ne donne donc pas un double accès à un service!!! En conséquence, lors de la duplication d'un lien, les services `utilisable_une_fois` se retrouvent `inutilisable` dans le lien dupliqué.

2.4.5 Destruction de liens

Un module a la possibilité de détruire des liens de son environnement. Dès que la demande de destruction est exécutée, le lien est inutilisable pour envoyer des messages, il ne peut non plus être placé dans des messages. Néanmoins la destruction effective du lien aura lieu à la fin de l'étape de traitement en cours pour

permettre au système de Transport de récupérer les noms uniques des modules destinataires des messages envoyés par les liens détruits ultérieurement à leur utilisation. Concrètement l'entrée de l'environnement passe dans l'état `a_detruire` lors de l'opération de destruction.

Naturellement, on ne peut pas détruire un lien placé dans un message.

```
FUNCTION DETRUIRE ( m : ID_MODULE ) : ID_ERREUR;

...

statut := DETRUIRE(m) ;
{ m devient inutilisable; le lien correspondant sera }
{ détruit à la fin de l'étape de traitement en cours }
{ statut signale une erreur si m n'est pas valide }
```

Primitive de destruction

En résumé, les structures utilisées dans ce chapitre pour permettre le nommage et la protection dans le système OMPHALE sont les suivantes. Elles permettent l'utilisation de noms locaux à chaque module (permettant ainsi le principe du moindre privilège) et le contrôle local des communications. Les opérations de gestion des liens offrent toute la souplesse nécessaire pour la mise en place de coopérations variées entre modules.

```
TYPE
     etat_service = (inutilisable,
                     utilisable_une_fois,
                     utilisable_n_fois,
                     deja utilise);
     liste_etat_service = array[ID_SERVICE]
                             of etat_service;
     lien = RECORD
              nom_unique : INTEGER;
              duplicable : BOOLEAN ;
              services : liste_etat_service
            END;
     etat = (vide, valide, dans_message, a_detruire);
     environnement = ARRAY [ID_MODULE] OF
                       RECORD
                         etat_entree : etat,
                         le_lien :lien
                       END;
```

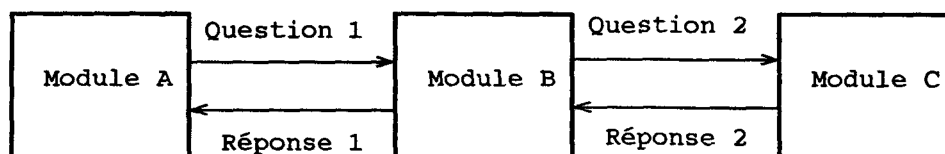
Cela se caractérise par l'introduction des primitives DUPLIQUER,

NON_DUPLICABLE, MODIFIER_ETAT_SERVICE, DETRUIRE.

2.4.6 Exemples d'utilisations des liens

2.4.6.1 Long Retour

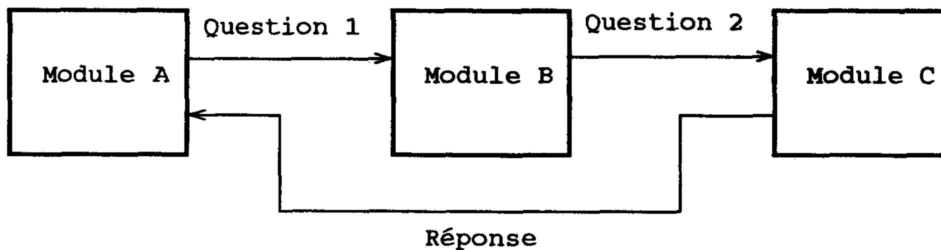
La technique du **Long Retour** [STR81] est particulièrement adaptée au cas des Questions–Réponses imbriquées: Pour répondre à une question du module A, le module B doit interroger le module C et attendre sa réponse.



Questions-Réponses imbriquées

Dans bien des cas qui relèvent de ce schéma, le module B ne fait que transmettre la question de A au module C, il ne fait pas lui–même de traitement. Son rôle est bien plutôt de choisir le bon module qui fournira la réponse à A. C'est alors un module "**gérant**" des modules pouvant répondre aux questions qui lui sont posées. Il y a deux sortes de telles "*gérances*": i) celle où les modules gérés ont tous des fonctionnalités spécifiques, dans ce cas le gérant est un "**dispatcher**" qui va choisir le module adapté à la question; ii) celle où les modules gérés ont tous les mêmes fonctionnalités, et dans ce cas, ce sont des critères de disponibilité, charge, etc... qui sont analysés par le gérant pour choisir le module.

Dans tous ces scénarios, il serait souhaitable que la réponse fournie par le module C ne repasse pas par le module B car elle ne le concerne pas. La technique du Long Retour permet cela: dans le cas d'appels (de type procédural) imbriqués, la technique du Long Retour évite l'attente des processus intermédiaires par transmission lors de l'appel du nom du processus émetteur de la question initiale.



Questions-Réponses imbriquées (long retour)

Cela est très facilement réalisable avec la communication asynchrone munie du transfert possible des noms (liens) de modules dans les messages.

```
MODULE A;
    ...
    ...
    WITH requete DO BEGIN
      question := ...
      service_reponse.destinaire := MYSELF;
         {transfert d'un lien vers A}
      service_reponse.service := reponse_a_la_question
    END;
    ecrit_message (B, accepter_question, requete);
    init_choix(liste); valide_choix(liste,reponse_a_la_question);
    mes:= ATTENDRE(liste)
    ...
    ... la réponse est arrivée
    ...
END;

MODULE B;
    ...
    ...
    init_choix(liste); valide_choix(liste,accepter_question);
    mes:= ATTENDRE(liste);
    ...
    ...  choix de C
    ...
    ecrit_message (C, traiter_question, mes.contenu);
         { on retransmet la requete    }
         { et le lien qu'elle contient }
    ...
    ...
END;

MODULE C;
```

```
    ...
    ...
    init_choix(liste); valide(liste, traiter_question);
    mes:= ATTENDRE(liste);
    requete:= mes.contenu;
    ...
    ... calcul de la réponse à requete.question
    ...
    ecrit_message (requete.service_reponse.destinataire,
          requete.service_reponse.service,
          reponse);
    ...
    ...
END;
```

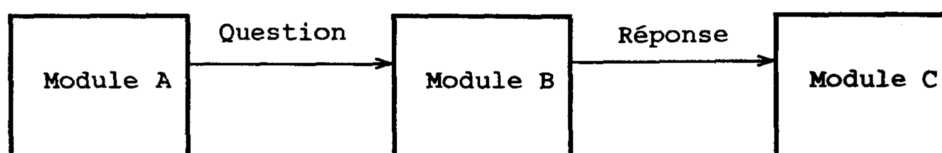
Technique du Long Retour

On voit que la disponibilité de B est globalement augmentée pour traiter les requêtes de réponse. C'est une technique intéressante dans le cadre d'architectures réparties permettant donc un certain parallélisme physique. On remarque aussi que c'est une technique qui n'est pas possible avec le pur RPC, qui s'avère être, dans ce cadre, un outil de "*trop haut*" niveau.

2.4.6.2 Continuations

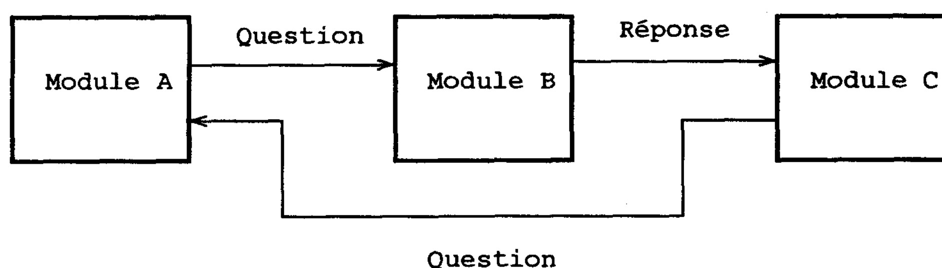
La notion de "**continuation**" est assez ancienne, elle correspond à fournir *explicitement*, lors d'un appel de sous-programme, l'adresse de retour (en général dans les systèmes sans continuation, l'adresse de retour est fournie *implicitement* sur la pile par le mécanisme d'appel).

Dans notre cadre de coopération de processus, cela signifie qu'un processus A va pouvoir fournir, lors d'une question à un module B, un service de réponse non pas chez lui-même, mais chez un module C.



Continuation

On remarquera que cette technique est plus primitive que la précédente, puisqu'il suffit de redessiner le schéma précédent en ajoutant un flèche Question entre C et A pour obtenir le Long Retour:



Continuation (long retour)

Comme indiqué ici, la continuation permet d'utiliser un module B comme "**filtre**" entre A et C, alors que B a été écrit pour satisfaire au protocole Question–Réponse. Cela n'est pourtant vrai que si B est écrit "*indépendamment des applications*" au sens du modèle Serveur du chapitre 1.

```
MODULE A;
   ...
   ...
   WITH requete DO BEGIN
     infos := ...
     service_reponse.destinaire := C;
        {transfert d'un lien vers C}
     service_reponse.service := recevoir_infos;
   END;
   ecrit_message(FILTRE, traiter_infos, requete);
   ...
   ...
END;


MODULE FILTRE;
   ...
   ...
   init_choix(liste); valide(liste, traiter_infos);
   mes:= ATTENDRE(liste);
   requete:= mes.contenu;
   ...
   ... traitement des infos; fabrique nouvelles_infos.
   ...
   ecrit_message(requete.service_reponse.destinataire,
         {utilise le lien vers C}
         requete.service_reponse.service,
         nouvelle_infos);
   ...
   ...
END;


MODULE C;
   ...
   ...
   init_choix(liste); valide_choix(liste,accepter_infos);
   mes:= ATTENDRE(liste);
   infos := mes.contenu; { il s'agit des nouvelles infos}
   ...
   ...
END;
```

Filtre par Continuations

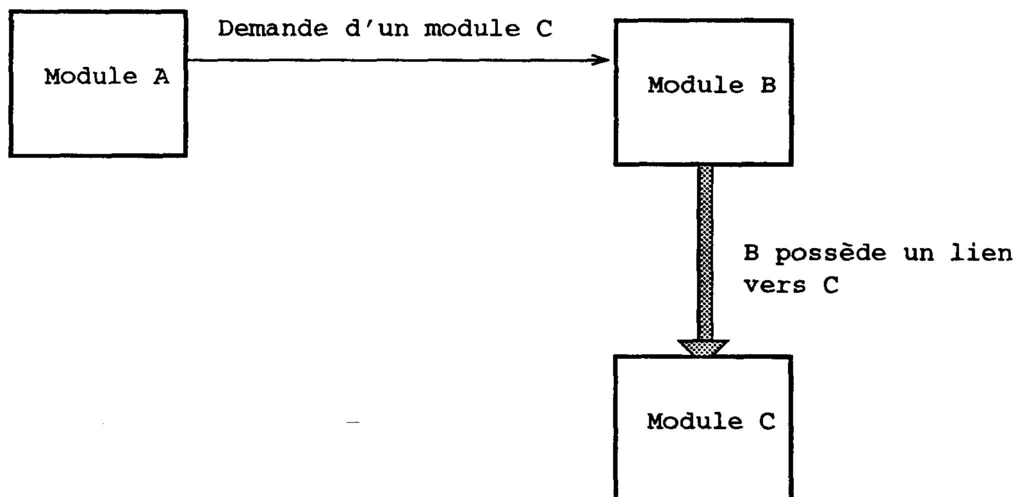
Cette technique est réalisable ici parce qu'il est possible de transmettre des noms (liens) de modules dans les messages. Cela offre, allié au mécanisme simple d'envoi asynchrone, toute liberté pour faire coopérer différentes entités.

2.4.6.3 Modules Gérants

Les modules gérants, déjà introduits dans cette section lors du long retour, sont ici compris au sens de [KIE78] [SIL77]. Dans ce cadre, un module gérant est un allocateur de ressources pour des modules demandeurs.

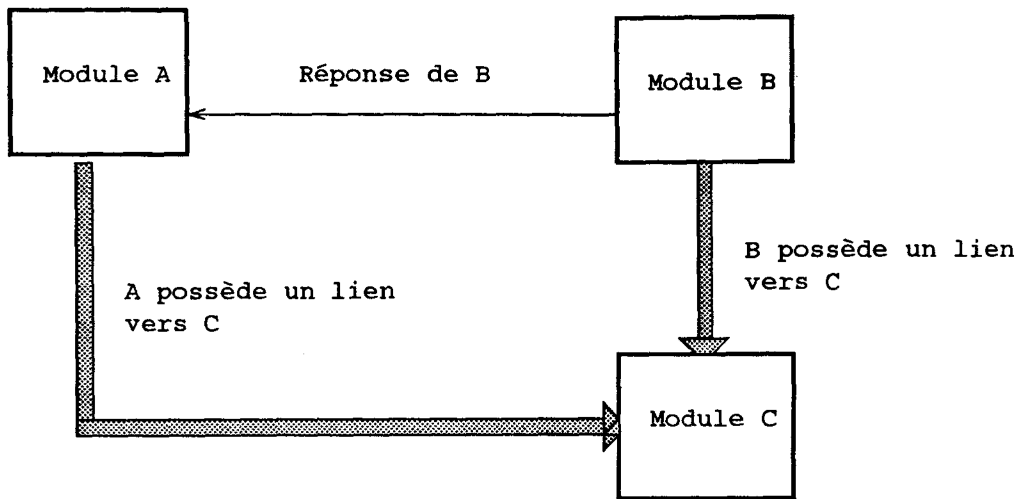
Lorsque la coopération d'un module A avec un module C doit être assez complexe, et que de plus la liaison A–C doit s'établir dynamiquement au vue de critères qui ne sont connus qu'à l'exécution, il n'est pas souhaitable qu'un module B (responsable de l'évaluation des critères et de l'établissement de la liaison) reste un intermédiaire de toute la coopération entre A et C.

Dans ce cas le module B doit être un allocateur du module C pour le module A. **Première phase**: Demande du module A au module B qui connaît les différents modules C pouvant être utilisés.



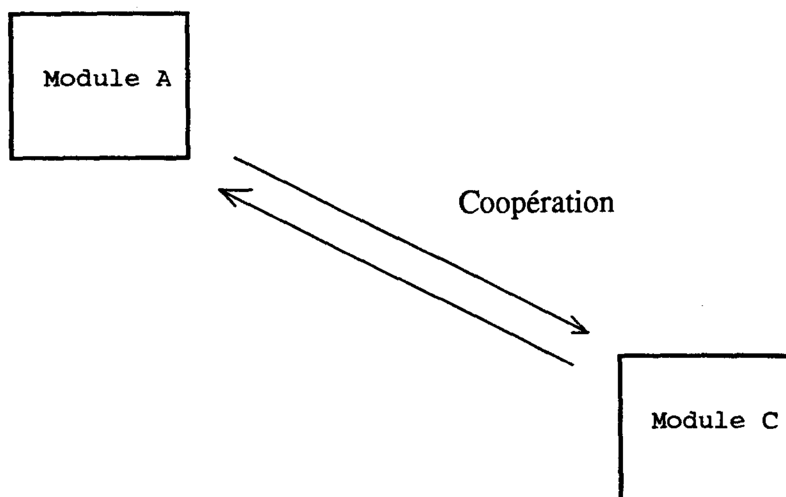
Première Phase

Deuxième phase: B répond en fournissant à A un lien vers le C choisi. Il va pour cela utiliser la duplication de liens et éventuellement restreindre (par modifier_etat_service) les droits de A sur C. En conséquence A possède maintenant un lien vers C.



Deuxième Phase

Troisième phase: A peut maintenant coopérer sans intermédiaire avec C.



Troisième Phase

Quatrième Phase: A doit signaler à B la fin de la coopération avec C pour que B

puisse allouer ce module à un autre demandeur. Il suffit ici que A détruise le lien vers C qu'il possède (ou le retourne à B) et signale cela à B.

```
MODULE A;
    ...
    ...
    WITH service_reponse DO BEGIN
      service_reponse.destinaire := MYSELF;
        {transfert d'un lien vers A}
      service_reponse.service := recevoir_nom;
    END;
    envoi(GERANT, allouer, service_reponse);
    init_choix(liste); valide_choix(liste, recevoir_nom);
    mes:= ATTENDRE (liste);  { on reçoit un lien vers C }
    C:= mes.contenu;
    ...
    ... Cooperation avec C
    ...
    envoi(GERANT, desallouer,C);
       {on retourne ici le lien vers C }
END;


MODULE GERANT
    ...
    ... gestion d'une table d'allocation
    ...
    init_choix(liste);
    valide(liste, allocation);
    valide(liste, desallocation);
    mes:= ATTENDRE(liste);
    CASE mes.adresse_service.service OF
       allocation:
          service_reponse:= mes.contenu;
          ...
          ... choix d'un module M; son état est mis à alloué.
          ...
          DUPLIQUER(M, C);
          envoi(service_reponse.destinataire,
                service_reponse.service,
                C);
       desallocation:
          module_libere:= mes.contenu;
          DETRUIRE(module_libere); {on détruit la copie}
          ...
          ... l'état du module est mis à libre
    END;
    ...
    ...
END;
```

Module Gérant et utilisation

2.5 Création et Destruction de modules

2.5.1 Création de modules

OMPHALE est un système dédié aux applications réparties structurées en termes de nombreux processus communiquants. Dans ce domaine, la création dynamique de modules est une opération fréquente. Elle est entreprise par un envoi d'une requête de création à un module Serveur de Modules. La requête précise un nom de fichier binaire.

Différents services peuvent être offerts dans un Serveur de Modules. On trouvera par exemple:

1) Service de Création locale

2) Service de Création sur un site choisi par le système d'exploitation.

3) Service de Création sur le site le moins chargé.

Le Serveur de Modules réalise la coopération entre le Serveur de Mémoire et le Serveur de Fichiers concernés pour préparer l'image mémoire du processus puis prévient le noyau pour la prise en charge du nouveau module (création du chaînon pour ce processus dans les listes de processus, création des zones de messages et de l'environnement...). Le module nouvellement créé est alors lancé pour exécution de son contrôleur.

La réponse finale du Serveur de Modules au module demandeur de la création est un message contenant un lien vers le module créé, permettant ainsi au module demandeur d'utiliser les services du module créé.

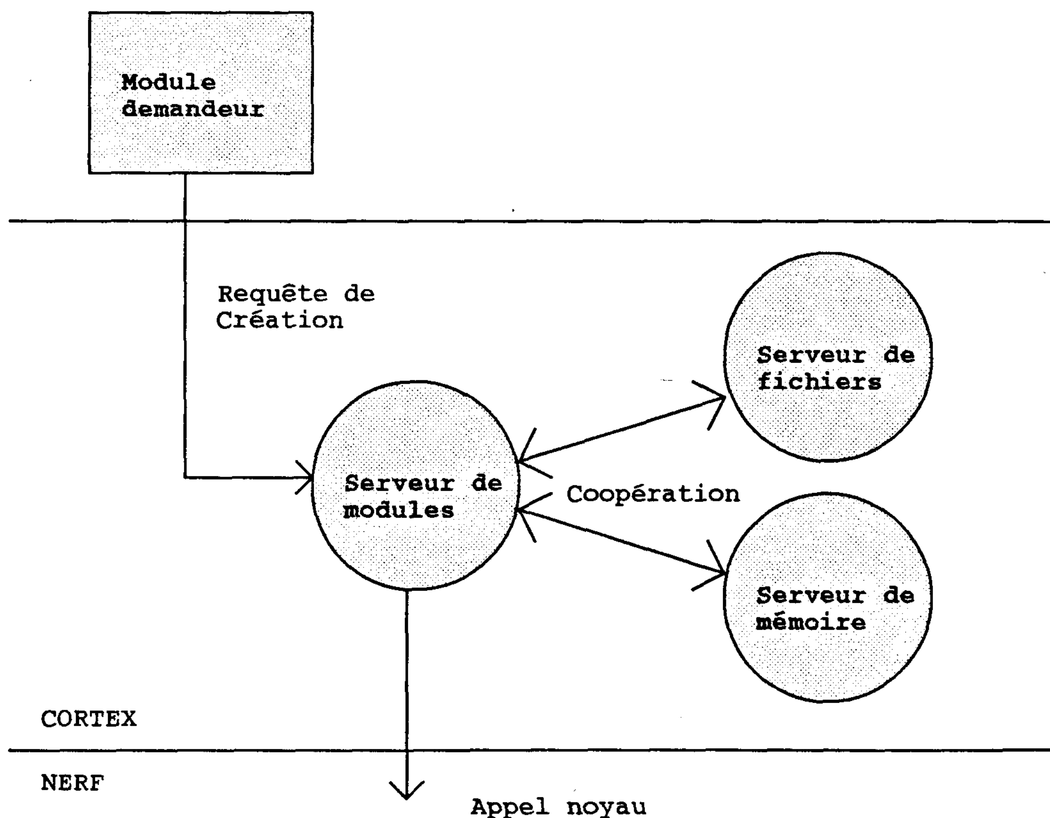
Le scénario suivant est typique d'une opération de création dynamique.

```
{ 1- Création du message de requête }
{ a un Serveur de Modules }
st := ecrit_message( mon_serveur,
                     creer, "/usr/OMPHALE/lib/tampon"
                     adresse_de_service_de_reponse);


{ 2- Envoi du message et attente de la réponse }
mes := ATTENDRE([reponse]);


{ 3- Récupération d'un nom utilisable pour le module créé}
{    mon_tampon est de type ID_MODULE }
mon_tampon := mes.contenu;


{ 4- Utilisation du module créé }
st := ecrire_message( mon_tampon, ecrire, infos)
```

2.5.2 Primitive de Création

L'appel du noyau pour la prise en charge et le démarrage d'un nouveau processus se fait par l'utilisation d'une primitive `creer_module`. Elle ne peut être entreprise que par un module Serveur de Modules après préparation de l'image mémoire.

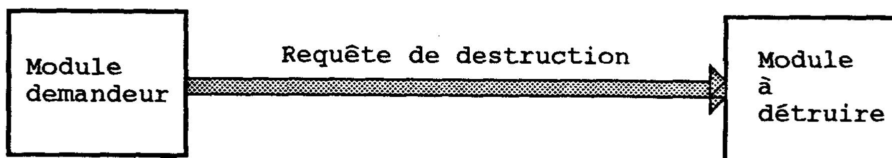
Le NERF va créer les Zones d'Emission et de Réception pour ce module. Il crée aussi l'environnement initial pour ce module. L'environnement initial d'un module comprend implicitement un lien vers le module lui-même. En général, la demande de création sera accompagnée de liens vers le Serveur de Noms et le Serveur de Modules utilisés par le module demandeur. Ces liens se retrouveront dans l'environnement initial du module créé. On réalise par ce mécanisme un *héritage* entre modules logiquement liés (héritage au sens de l'héritage des descripteurs lors du "*fork*" d'UNIX).



2.5.3 Destruction de module

2.5.3.1 Service de destruction

Un module peut être détruit par un autre module par envoi de message. Le module demandeur de la destruction envoie un message de requête de destruction au module à détruire. Le traitement par ce dernier du message consiste à se terminer (par utilisation d'une primitive spécifique du noyau).



Destruction par message

Ce mécanisme de destruction est soumis aux schémas d'exécution, de communication et de protection du système OMPHALE, c'est-à-dire:

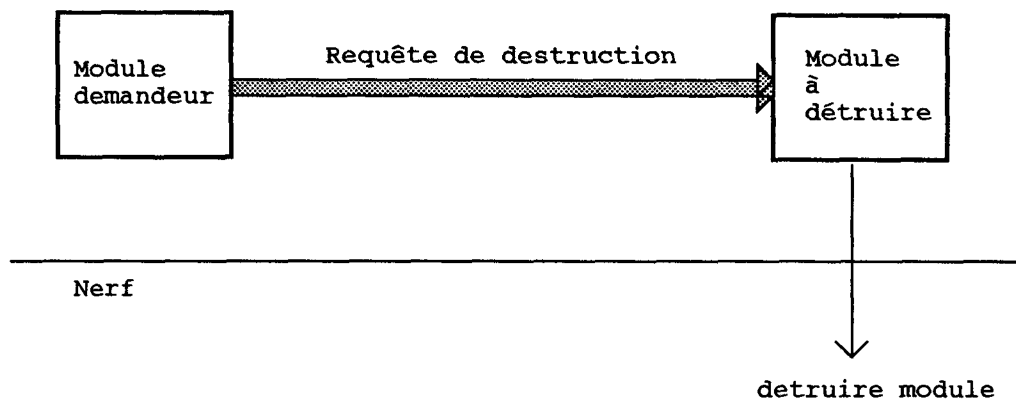
1) Le module à détruire n'est pas interrompu. La destruction aura lieu lorsque le module sera réveillé pour traiter le requête de destruction. Cela signifie que le module **peut refuser** la destruction. En général cependant, un tel service sera toujours validé avec une priorité importante.

2) Un module demandeur de destruction ne peut envoyer la requête que s'il possède un lien vers le module à détruire, de plus ce lien doit autoriser le service de destruction.

3) le protocole de destruction (contenu du message de requête, acquittement éventuel, traitement complexe d'une requête de destruction...) est construit au niveau application; le système lui-même ne faisant que transporter le message de destruction.

Un tel mécanisme simple permet d'implanter tout scénario de terminaison normale d'une application par le simple mécanisme d'envoi de message. Un module peut, par exemple, demander la destruction d'autres modules (si l'application l'exige) avant de se détruire lui-même. On retrouve ici l'intérêt d'un mécanisme construit au-dessus du noyau (et donc souple) à comparer avec un mécanisme (rigide) construit dans le noyau.

La destruction effective du module (qui traite une requête de destruction) se fait par une primitive `detruire_module`. Cette primitive détruit le module qui l'exécute. La Zone d'Emission du module à détruire est passée au Système de Transport pour envoi des messages qui s'y trouvent; l'Environnement est détruit par destruction des liens qui s'y trouvent; la Zone de Réception est détruite, tous les messages qui s'y trouvent sont perdus. Le noyau prévient le Serveur de Modules et le Serveur de Mémoire de la récupération de l'espace alloué au module.



Destruction par message

2.5.3.2 Destruction immédiate

Chaque module est muni d'un service implicite particulier de destruction immédiate. Si, pour les autres modules, il s'agit d'un service comme les autres (requête par envoi de message, nécessité d'avoir un lien autorisant ce service...) sa prise en compte échappe cependant au schéma normal d'exécution: lorsque le Système de Transport dépose un message dans la file d'attente de ce service de la Zone de Réception d'un module, il avertit le noyau que ce module doit être détruit, le noyau entreprend alors la destruction immédiate du module sans attendre le passage en attente du module. Les messages de la Zone d'Emission du module ne sont pas envoyés car il s'agit ici d'interrompre l'activité du module.

Ce type de destruction est un mécanisme utilisé au niveau de la gestion système permettant de *corriger* le système à la suite de pannes.

La destruction immédiate d'un module est aussi entreprise lorsqu'un module passe en attente en ne validant aucun service. Il passe donc en attente *infinie* . Le noyau détecte ce fait au moment du passage en attente et entreprend la destruction du

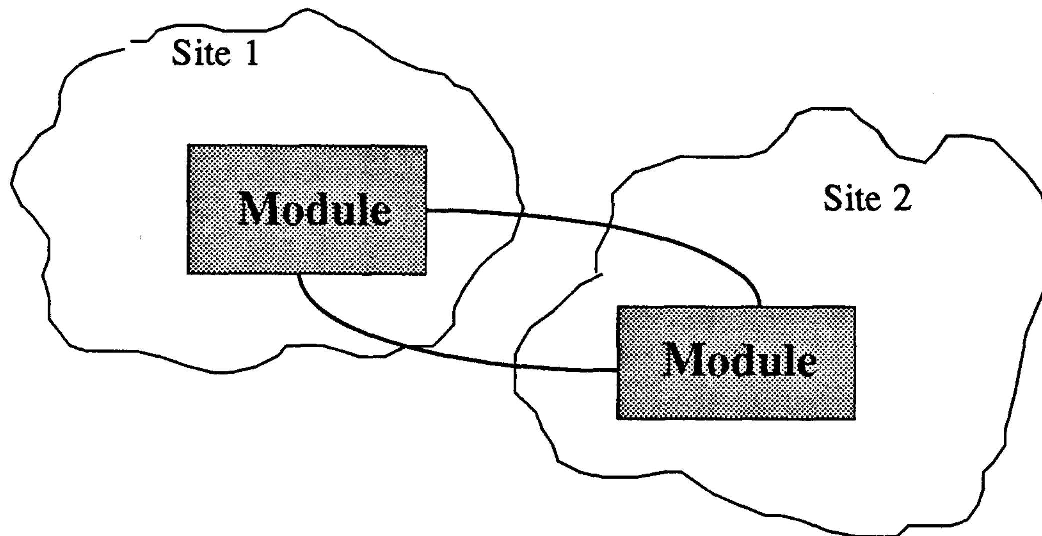
module. Le module n'est pas ici interrompu car il est en attente mais il ne se réveillera pas. La primitive `detruire_module` et celle de mise en attente sans valider de services sont en fait équivalentes.

2.5.3.3 Problèmes liés à la destruction

1) Dans les systèmes à capacités, il est nécessaire de se protéger des capacités qui *"pointent dans le vide"*, c'est-à-dire pour lesquelles il n'y a plus d'objet valide à l'adresse contenue dans la capacité. Dans le système OMPHALE, les liens sont des capacités mais i) qui ne pointent que vers des modules et ii) qui ne servent qu'à envoyer des messages. Le problème des liens "dans le vide" se pose différemment:

Que se passe-t-il lorsqu'un module A garde un lien vers un module B qui a été détruit ? Le module A va toujours pouvoir envoyer des messages vers le module détruit, le Système de Transport va acheminer le message vers le site concerné, mais le NERF local ne pourra le délivrer au module. Le message sera alors simplement perdu. Il n'y a pas nécessité de mettre en place des mécanismes de bas niveau pour se prémunir de ces pertes car le système complet est fondé sur un mode d'envoi de message sans acquittement et donc les applications doivent toujours être programmées de façon défensive vis-à-vis de la perte de message (essentiellement par "échéance").

2) Certains modules peuvent devenir *"non référençables"*. Lorsqu'il n'existe plus de lien vers un module A dispersé dans les autre modules et que le module A est en attente, il ne pourra plus se réveiller et personne ne pourra le détruire. On peut de la même façon trouver des boucles de références *"fantômes"* qui n'ont plus de raison d'exister car plus aucune interaction avec le reste du système ne peut survenir. Le problème est alors un problème de **récupération de ressources** : il faut récupérer la mémoire allouée à ces modules. La solution passe par un mécanisme de *"ramasse-miettes"* ou *"garbage collector"* qui parcours le graphe (réparti !!) des références à la recherche des modules fantômes. En toute généralité, la migration possible des modules d'un site à l'autre rend le ramasse-miettes des boucles de références difficile : il faut interrompre toutes les autres activités du système. Ce ramasse-miettes général ne peut donc être entrepris qu'à des moments privilégiés permettant l'arrêt des activités.



Boucle fantôme répartie

Le problème est néanmoins simplifié lorsqu'il n'existe pas de migration de modules. Un algorithme basé sur le comptage de références et sur la propagation de marques permet de réaliser le ramasse-miettes comme une tâche "*arrière*" du noyau sans interrompre le système. La version SPS7 du noyau OMPHALE ne réalise que le comptage des références pour chaque module : les **compteurs de liens** sont incrémentés ou décrémentés à chaque opération de création/duplication et destruction de liens. Lorsqu'un module est en attente sans échéance, et que son compteur de référence est à zéro, il est détruit.

2.6 Conclusion sur la description de NERF

Le noyau NERF décrit ici, est le résultat de l'utilisation du Modèle Serveur comme base d'un exécutif réparti. NERF supporte la création, coopération et destruction de serveurs sur un ensemble de site.

A partir du Modèle Serveur, nous avons donc

1) implanté la notion de requête de service par l'utilisation d'un mécanisme de transfert asynchrone de message uniforme sur l'ensemble des sites. L'espace de stockage des messages en cours de transfert est découpé en deux espaces distincts: Un appartenant au contexte de l'émetteur du message, l'autre dans le contexte du récepteur. Cela offre les avantages, pour le premier, de pouvoir réaliser un contrôle global des messages avant transfert, et pour le deuxième de pouvoir réaliser facilement un mécanisme d'attente sélective de messages.

2) implanté la notion de contrôleur par l'utilisation d'une primitive unique d'envoi–attente–réception . Un serveur paramètre localement cette primitive (en validant des services), il s'impose ainsi un fonctionnement en automate qui assure la cohérence de son état.

3) fournit les possibilités de définition de serveurs autonomes réutilisables par un mécanisme de nommage permettant le transfert de noms de serveurs dans les messages. L'utilisation de noms locaux et de liens permet de plus de fournir un domaine de protection au niveau du service.

L'ensemble des mécanismes décrits ici permettent la réalisation d'un noyau simple et uniforme : un seul type d'entité, un seul mode de communication valable pour toute entité. Etant à la fois support du modèle Serveur et d'exécutions réparties, NERF est une base qui nous semble prometteuse, à la fois pour la prise en charge des Langages Orientés Objets Parallèles et pour la création de Systèmes d'Exploitation Répartis d'Objets.

Dans le système d'exploitation réparti OMPHALE, les grandes fonctionnalités systèmes vont être réalisée par des serveurs s'exécutant au dessus du noyau NERF. Nous en donnons maintenant un aperçu. On lira à ce sujet la thèse de Jean-François Méhaut en préparation.

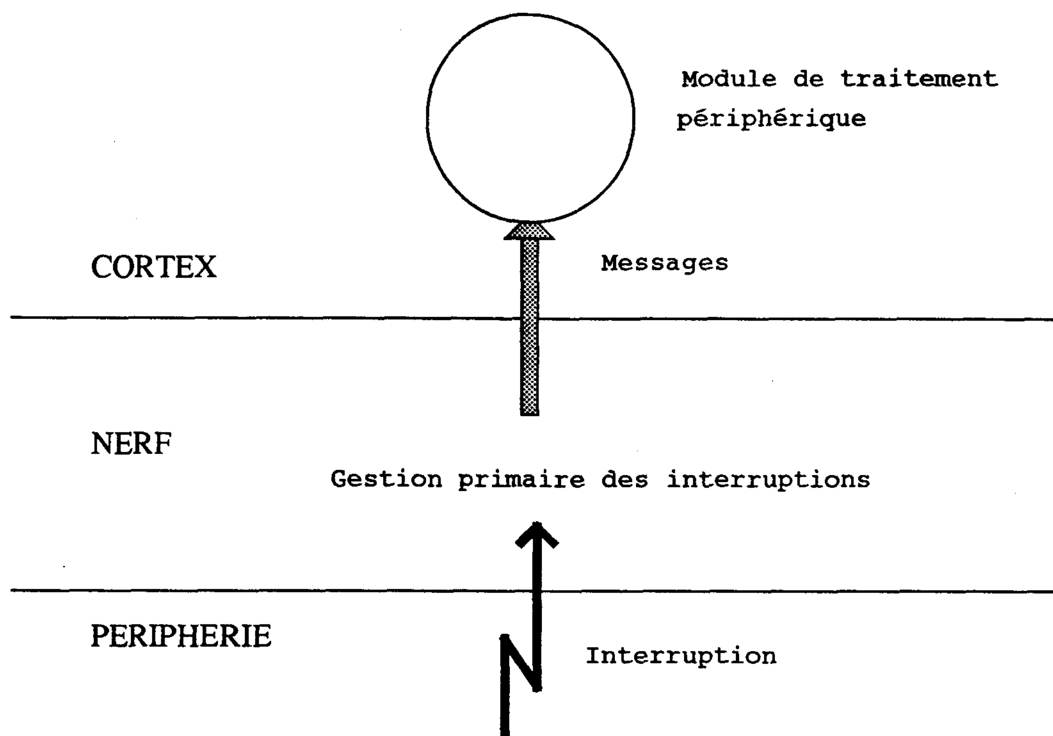
2.7 Aperçu de la couche CORTEX

OMPHALE est un système à noyau réduit. La plupart des tâches importantes d'un système d'exploitation vont être entreprises au–dessus du noyau par des modules serveurs qui obéissent au même schéma d'exécution que les modules utilisateurs. Cette couche de modules systèmes est actuellement à l'étude sur une plate-forme expérimentale du noyau NERF. On ne fait ici que décrire succinctement les fonctionnalités voulues.

Les différentes tâches pouvant être réalisées par ces serveurs sont:

1) **Gestion des périphériques**:
La gestion des périphériques peut être entreprise au–dessus du noyau en ne laissant au noyau que le travail primaire de gestion des interruptions, c'est–à–dire en général le dialogue de très bas niveau avec l'organe périphérique et la stockage dans des tampons de données. C'est le travail habituel des conducteurs ("*drivers*") de périphériques. Il y a ici un problème de performance du système vis–à–vis de périphériques à haut débit.

Le travail secondaire (gestion d'informations systèmes liées à un périphérique) peut, quant à lui, être entrepris par des modules de la couche CORTEX sollicités par le noyau (qui leur envoie des messages). Ce travail est alors "*hors noyau*" et peut donc être facilement modifié, adapté... face à des changements systèmes ou périphériques.



2) **Gestion du transport de messages inter–site**:

Comme on l'a vu le transport de message inter–site est entrepris entre deux NERFs distants par un module réparti appelé Station de Transport. Cette Station de Transport communique par messages avec les NERFs du système, le contenu des messages étant les messages à transmettre. Cette Station encapsule le support physique de communication dont les protocoles d'utilisation ne sont donc même pas connus du noyau lui–même.

Alors que le mode de communication (d'un point de vue module) est le mode **datagram** [TAN81b], c'est à dire le simple envoi de message sans acquittement ni assurance de délivrance; à l'intérieur de la Station de Transport, le mode de communication peut être plus sophistiqué (stream [SUN86b] ...). Alors que le transfert intra–site peut être rendu fiable par validation des logiciels systèmes, le transfert inter–sites a la fiabilité du support de communication et la Station de Transport peut utiliser différents protocoles de transfert (réessai, contrôles...) permettant de fournir la fiabilité voulue.

La Station de Transport est formée d'un module par site. Ces modules coopèrent, créant ainsi virtuellement le module réparti de Transport. La Station de transport n'est pas directement utilisée par les modules mais fait partie du Système de Transport qui permet le transfert de messages de NERF à NERF.

3) **Gestion de la mémoire centrale et la mémoire de masse**:
Comme le fait Minix [TAN87], la mémoire centrale et la mémoire de masse peuvent être gérées au dessus du noyau par l'intermédiaire de processus serveurs (Serveur de mémoire et Serveur de fichiers).

Un Serveur de mémoire (il y en a un par site) répond aux demandes d'allocation et de désallocation de mémoire sur le site. Ces demandes peuvent venir du noyau ou d'autres modules, du site ou d'autres sites.

Un Serveur de Fichiers répond aux demandes d'ouverture, fermeture, lecture, écriture... de fichiers pour la mémoire de masse (disques magnétiques ou autres) qu'il gère. Comme le fait Chorus , la coopération de différents Serveurs de Fichiers est une manière de créer un Système de Fichiers Réparti permettant d'utiliser globalement et de manière transparente différents espaces disques physiquement distants.

3) **Gestion d'une couche de nommage symbolique des modules**:
La couche CORTEX gère un espace de noms symboliques grâce à des modules spécialisés appelés **Serveurs de Noms.**

Un serveur de noms est un module qui encapsule une liste d'association (nom symbolique, lien). Le principal service qu'il rend est de fournir un lien à partir d'un nom symbolique.

Le scénario suivant est typique d'une opération de récupération d'un lien par un nom symbolique.

L'intérêt des Serveurs de Noms est de permettre l'établissement dynamique des liaisons entre modules. Cela est particulièrement intéressant dans les cas suivants:

— **Utilisation d'une ressource du système**:
En prenant le cas d'une imprimante, il s'agit alors d'établir la connexion avec le module gérant de l'imprimante. Il est préférable d'établir la connexion au moment voulu de l'exécution en utilisant un nom symbolique plutôt que de "*souder*" le module imprimante dès la compilation en créant un lien dans l'environnement initial du module utilisateur. La méthode dynamique permet de plus de choisir à l'exécution entre plusieurs modules équivalents celui qui sera utilisé en s'adaptant à des problèmes de charge ou de pannes. Pour cela les noms symboliques peuvent être soit complets pour désigner un module précis, soit incomplets (dans le sens des "star–conventions" d'Unix) pour désigner un module quelconque parmi un ensemble de modules pouvant rendre les mêmes services.

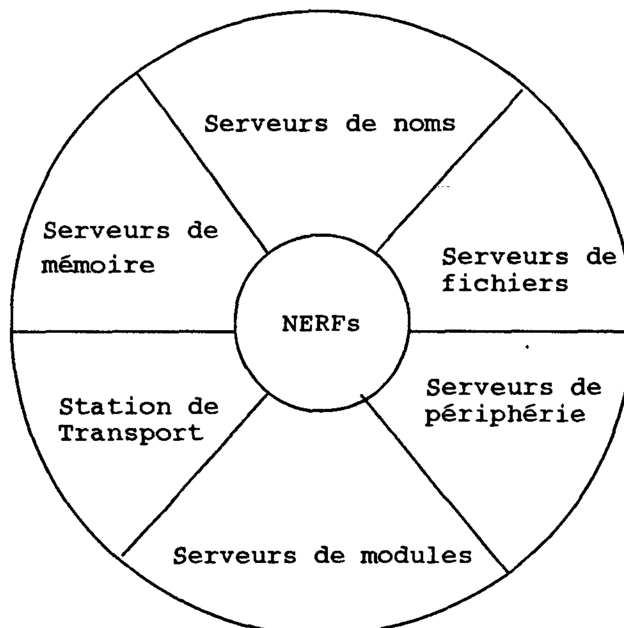
— **Etablissement de liaisons croisées**:
Lorsque deux modules veulent communiquer par question–réponse, il est nécessaire que chaque module possède un lien vers l'autre. Ces liens peuvent être établis à la compilation si les modules sont étroitement liés dans l'application , mais, dans un cadre plus général, il est obligatoire que ces modules se connaissent au départ par des noms symboliques et utilisent un Serveur de Noms pour obtenir

les liens permettant de communiquer.

Un Serveur de Noms offre, outre le service d'établissement d'une liaison, des services d'enregistrement et de destruction d'associations nom symbolique–lien.

En général, un module héritera au moment de sa création d'un lien vers un Serveur de Noms connu de son père, retrouvant par ce mécanisme un environnement de travail équivalent à celui du père. Un serveur de nom initial permet le démarrage de ce mécanisme.

4) **Gestion de la Création et de la Destruction des modules**:
La création et la destruction de modules, qui sont décrits plus haut, fait classiquement intervenir les Serveurs de mémoire (pour allouer et désallouer l'espace mémoire nécessaire aux processus), les Serveurs de Fichiers (pour obtenir les images mémoires des processus) et le noyau pour la prise en compte interne des opérations de création et de destruction des modules. Des modules particuliers du CORTEX appelés Serveurs de Modules se chargent de réaliser la coopération de ces entités. Et c'est alors par un simple envoi d'une requête de création à un Serveur de Modules qu'un module va pouvoir demander la création d'un autre module. Selon que le Serveur de Modules impliqué se trouve sur le site du demandeur ou sur un site distant, la création se fera sur le site local ou distant.Les modules Serveurs de Modules sont le lieu naturel de la gestion de la régulation de charge sur l'ensemble des sites. Une régulation automatique de charge étant entreprise par communication d'informations de charge entre Serveurs deModules.



La couche CORTEX

Chapitre 3
Deux réalisations du noyau OMPHALE

Deux réalisations du noyau OMPHALE ont été entreprises. L'une sur une architecture classique formée de machines SPS7–UNIX reliées par ETHERNET, l'autre sur une architecture spécifique développée pour le support du système OMPHALE.

Le propos de la première réalisation est de montrer la faisabilité du noyau NERF et d'expérimenter dans la réalisation de la couche CORTEX. Le propos de la deuxième réalisation est tout autre: il s'agit ici d'expérimenter dans le domaine des machines à changement de contexte rapide, et cela grâce à des dispositifs matériels appropriés. Dans ce domaine nous avons construit une machine expérimentale correspondant à un site (appelé Site 0) d'une architecture de système OMPHALE. Il s'agit d'une machine bi–processeur, l'un exécutant le noyau NERF, l'autre les couches au dessus. Un prototype de NERF tourne sur cette machine. Jean–Marie Place travaille actuellement à des mesures de performances.

3.1 Architecture SPS7–SPART–ETHERNET

Le noyau NERF et des prototypes du CORTEX ont été développés sur des machines multiprocesseurs SPS7 de BULL reliées par un réseau local ETHERNET [MET76] . Le système de développement est SPIX version BULL d'UNIX System V [BUL85a], l'exécutif a été construit au dessus du noyau temps réel SPART [BUL85b].

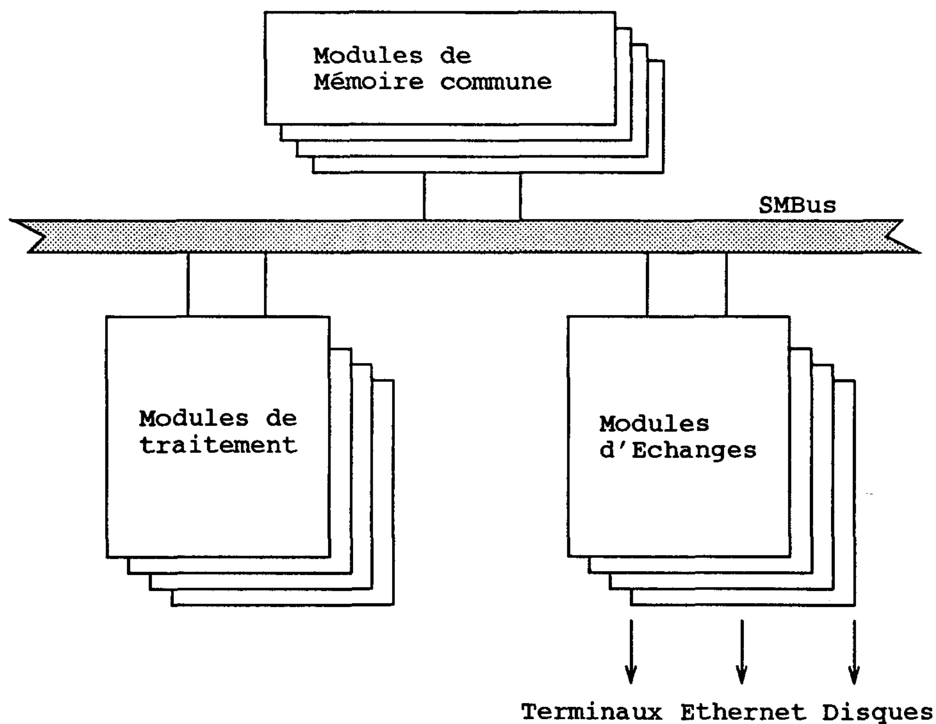
3.1.1 L'architecture matérielle SPS7

La machine SPS7 de BULL est issue du projet SM90 du C.N.E.T. [FIN81]. C'est une architecture modulaire construite autour d'un bus particulier appelé SMBus. Le SMbus peut recevoir des Modules de Traitement (MT) ou/et des Modules d'Echanges (ME) (Module d'échange Ethernet, Module d'échange disque, Module de Mémoire globale, Modules de voies asynchrones, ...). Les conflits

d'accès au SMBus sont résolus par un mécanisme matériel d'arbitrage présent sur chaque module et repose sur la notion de jeton circulant.

Un module de Traitement comprend un processeur 680xx et de la mémoire privée ainsi qu'un bus local et de la mémoire locale. On peut avoir jusqu'à 8 MT sur le SMBus. Un Module de Traitement envoie des requêtes vers les Modules d'Echanges, cela permet de porter une part importante du travail sur les ME (eux-mêmes bâtis sur des processeurs de la catégorie 680xx).

Typiquement une SPS7 Unix simple comprend un Module de Traitement qui exécute Unix et les applications, un Module d'Echange Disque pour gérer la mémoire de masse et un dérouleur de cartouches, et un Module d'Echanges Asynchrones pour accueillir des terminaux.



Architecture modulaire de la SPS7

3.1.1.1 Mémoires

Chaque Module de Traitement est muni d'une Unité de Gestion Mémoire (UGM) qui

1) réalise la translation des adresses logiques dans la mémoire physique

2) permet la protection de la mémoire : type d'accès (lecture, écriture), mode d'accès (système, utilisateur), séparation entre les utilisateurs.

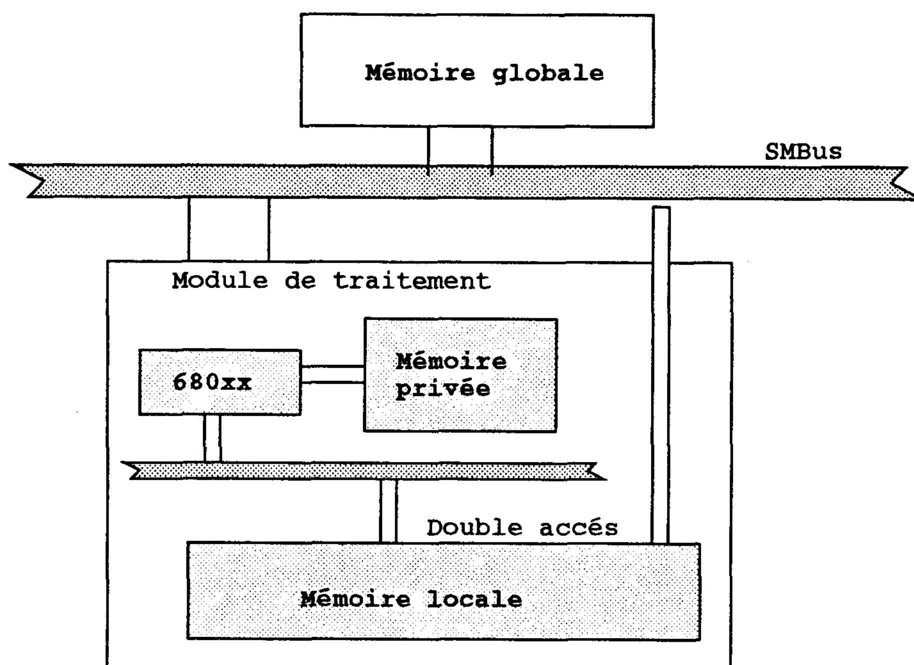
La mémoire de l'architecture est formée de trois niveaux :

1) **Mémoire privée d'un Module de Traitement**:
Elle n'est accessible qu'au Module de Traitement. Elle est accédée directement par le processeur du MT.

2) **Mémoire locale d'un Module de Traitement**:
Elle constitue typiquement la mémoire de travail du MT. Elle est située sur des cartes spécifiques connectées au bus local du MT. Elle peut être à **double accès** et donc connectée aussi sur le SMbus. Dans le cas du double accès, il faut choisir le type d'accès par configuration physique de la carte. Lorsque l'accès est configuré pour se faire par le SMbus, l'accès local n'est plus possible, et inversement.

3) **Mémoire globale**:
Elle est accessible par tous les Modules via le SMBus. Elle se trouve sur une ou plusieurs cartes spécifiques directement connectées sur le SMbus.

La communication (par partage de mémoire) entre les Modules de Traitement peut donc se faire soit par la mémoire globale soit par une partie de la mémoire locale utilisée à partir du SMbus. C'est la deuxième solution qui a été choisie dans notre configuration.

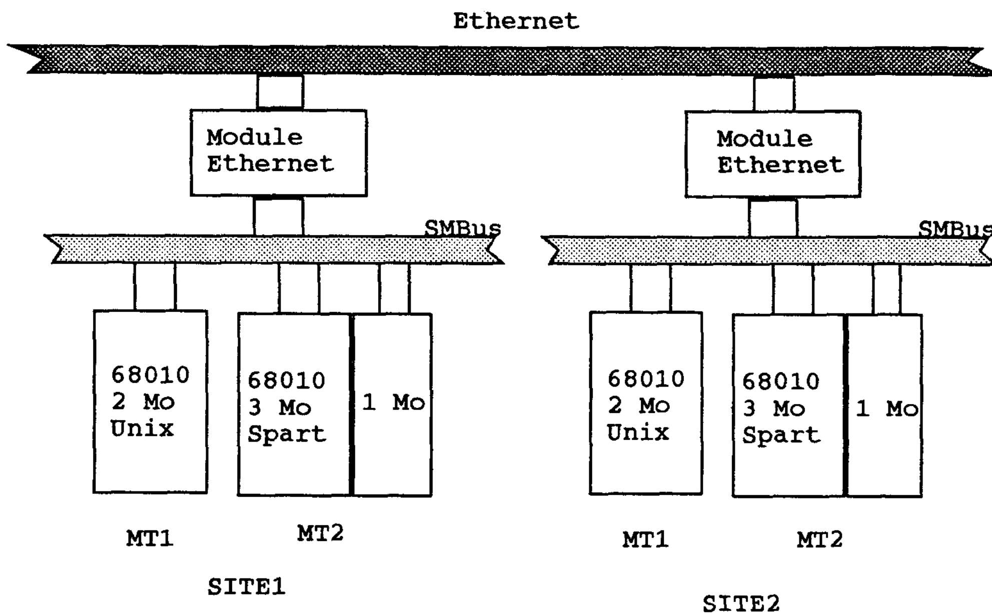


Trois niveaux de mémoire

3.1.1.2 Configuration matérielle utilisée

La configuration matérielle utilisée pour implanter le système OMPHALE est formée de deux SPS7 composées chacunes de deux Modules de Traitement. Sur chaque machine un MT exécute Unix avec deux Méga–octets de mémoire locale et un MT exécute SPART avec quatre Méga–octets de mémoire centrale dont un Méga–octet configuré en accès SMbus et donc jouant le rôle de mémoire globale. Chaque machine est munie d'un Module d'Echange Ethernet, un Module d'Echange Disque, un Module d'Echange Asynchrone. Il n'y a pas de carte Module de mémoire globale, c'est une partie connectée au SMBus des mémoires locales qui joue le rôle de mémoire globale.

Les modules de traitement exécutant Unix servent dans notre configuration au développement du système OMPHALE, ainsi que de support ETHERNET, les Modules SPART, quant à eux, exécutent le système créé.



Configuration matérielle

3.1.2 L'architecture virtuelle SPART

SPART est un noyau temps réel développé par BULL pour ses machines SPS7. Un noyau temps réel est un ensemble d'opérations primitives permettant la création et la gestion de processus, il offre de plus des possibilités liées aux contraintes typiquement temps réel. SPART présente différents moyens de synchronisation et de communication des processus. Ce noyau reprend les spécifications du noyau SCEPTRE [SCE82] [BRO84] du Bureau de Normalisation Internationale pour les noyaux temps réel. Les principales caractéristiques en sont :

1) **Interface indépendante de la machine physique**:
Les primitives SPART cachent les particularités de la machine pour ne montrer que des opérations conceptuelles pour la gestion du système. SPART crée donc une machine virtuelle SPART sur laquelle on va pouvoir construire un système d'exécution complet. Le noyau SPART et le système construit au-dessus sont structurés en terme d'**objets**. A chaque objet sont associées les opérations permettant de le manipuler. C'est une approche (type abstrait, orientée objet) qui a fait ses preuves pour la définition des systèmes complexes et qui est reprise ici.

2) **Extensibilité du noyau**:
Il est possible, tout en gardant la fiabilité et l'efficacité du noyau, d'ajouter des primitives au noyau. Cela se fait par la définition de nouveaux objets à gérer et des opérations permettant de les gérer.

3.1.2.1 Agences

Une agence est une unité fonctionnelle de SPART fournissant une ensemble d'opérations permettant de gérer un ensemble d'objets du même type (on pourrait dire une "*classe d'objet*"). Les objets gérés sont encapsulés dans l'agence, les opérations d'une agence sont des primitives qui s'exécutent en mode système, et il est donc impossible à un programme d'utiliser de façon illicite ces objets. Une agence est donc un moyen d'accéder à une ressource de la machine virtuelle SPART.

Il est possible de créer de nouvelles agences ayant toutes les propriétés des agences prédéfinies et donc d'augmenter les ressources accessibles de la machine virtuelle. La programmation d'une agence se fait en utilisant des primitives internes au noyau SPART qui ne sont pas accessibles aux programmes utilisateurs. L'interface pour créer des programmes utilisateurs au dessus de SPART se fait donc généralement par dialogue avec des agences.

Les requêtes de création fournissent en retour un "*nom système*" de l'objet créé. C'est ce nom qui est ensuite utilisé pour désigner l'objet dans un appel de primitive. Il est ainsi impossible d'accéder directement aux objets créés sans passer par les primitives autorisées.

On trouve les agences predefinies, Tâches, Programmes, Segments, Boîtes aux lettres, Sémaphores, Délais, Blocs d'entrées–sorties. De niveau plus élevé, on trouve aussi l'agence FMS (File Manager System) de gestion de fichiers.

3.1.2.2 Tâches

Une tâche correspond à un entité active qui exécute séquentiellement une suite d'instructions. Une tâche est créée dynamiquement à partir d'un programme (qui a été chargé par le service LOAD de l'agence Programme) puis lancée pour exécution (services CREATE_TASK et START_TASK de l'agence Tâche).
Une tâche est créée localement sur un module de traitement et y restera jusqu'à sa fin . On n'a donc pas ici de possibilité de migration.

Les tâches SPART peuvent coopérer grâce aux outils fournis par les agences Evénement, Sémaphores, Boîtes aux lettres ou Délais. Il est possible de créer de nouvelles agences installant un nouveau type de coopération entre tâches en combinant les mécanismes de base.

3.1.2.3 Synchronisation

SPART permet de gérer de la mémoire partagée entre tâches (et en particulier les Modules de mémoire globale de l'architecture matérielle). Une agence Segments permet l'allocation dynamique de mémoire, une agence Catalogue permet de conserver sous forme externe des noms systèmes obtenus dynamiquement. Ce

service permet donc le partage par récupération de noms systèmes du catalogue.

Lorsqu'il y a partage, il faut synchroniser les tâches coopérantes. Les outils fournis sont les **événements** et les **sémaphores**.

3.1.2.4 Communication

Les tâches SPART peuvent communiquer par messages. Ce mécanisme est installé par l'agence Boîtes aux lettres qui fournit les services CREATE_MBX, DELETE_MBX, SEND et RECEIVE. Les messages échangés ont une taille fixe de 8 octets. Les boîtes aux lettres ont une taille (en nombre de messages) fixée à leur création.

L'échange de messages peut se faire sur le même Module de Traitement ou entre Modules de Traitement selon que la boîte aux lettres se trouve en mémoire locale ou commune.

La réception de message peut se faire avec attente (lorsque la boîte est vide) ou sans attente. De même l'émission peut se faire avec attente (lorsque la boîte est pleine) ou sans attente.

3.1.3 ETHERNET

Les deux machines SPS7 font partie du réseau local installé au Laboratoire d'Informatique Fondamentale de Lille. Ce réseau ETHERNET est formé d'un serveur disque (SUN 3/260), d'un serveur d'impression (imprimante laser IMAGEN), de stations de travail (SUN, HP, Apollo) et de sous réseaux PC et Macintosh. Les SPS7 s'intègrent dans ce réseau par les possibilités de messagerie, de connexion à distance et de transfert de fichiers au-dessus des protocoles TCP/IP [TAN81b].

ETHERNET apporte un support de communication fiable et de grande vitesse (10 Mégabit/s). Les outils utilisables sous UNIX de type "**socket**" [SUN86b] fournissent au niveau programmation des possibilités de transfert d'informations de processus à processus entre machines.

SPART offre une agence ETHERNET permettant le transfert d'informations entre tâches SPART situées sur des machines différentes. Malheureusement cette agence n'a jamais pu fonctionner sur notre configuration et il a fallu utiliser les Modules de Traitement UNIX pour réaliser la communication par ETHERNET. A cette fin, il a donc fallu mettre en place la communication entre SPART et UNIX, ce qui a été réalisé par un mécanisme de boîtes aux lettres communes aux deux systèmes SPART et SPIX.

3.1.4 Agences spécifiques OMPHALE

3.1.4.1 Agence Messages

OMPHALE est un système où la communication inter-processus se réalise par messages. Un module OMPHALE manipule des messages par l'intermédiaire d'une Zone d'Emission. Les messages ont une structure propre liée au système OMPHALE qui ne rentre pas dans le cadre des messages SPART formés simplement de 8 octets. Le choix a été fait de créer une agence spécifique de messages OMPHALE. Les primitives fournies par cette agence sont les opérations prévues du système OMPHALE pour la manipulation de la Zone d'Emission d'un module. Les identifieurs de messages utilisés sont les noms systèmes retournés par SPART lors de la création d'objet SPART.

Chaque module manipule donc sa Zone d'Emission par l'intermédiaire des primitives de l'agence Messages. On y trouve les opérations:

1) de création, pour créer un message à partir de ses constituants et le placer dans la Zone d'Emission. Cela retourne un identifieur de message.

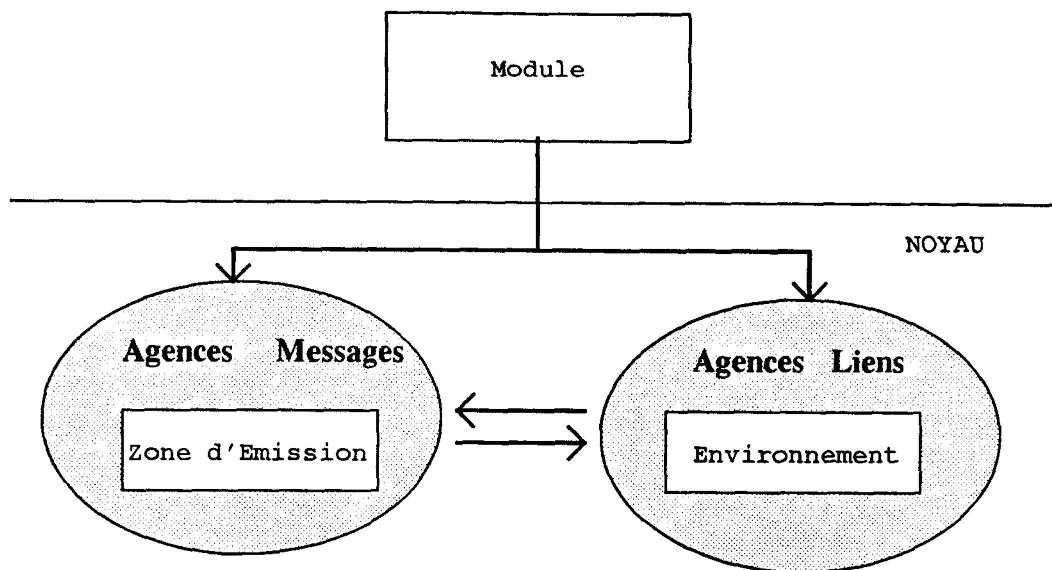
2) de destruction, pour détruire un message désigné par un identifieur.

Ces primitives, qui s'exécutent en mode système, retrouvent dans le contexte de la tâche appelante la localisation de la Zone d'Emission du module. Cette Zone est donc complètement cachée derrière ces primitives.

3.1.4.2 Agence Liens

Le contrôle des interactions entre modules se fait par l'intermédiaire de Liens qui renferment les droits des modules. Un lien est donc une structure qui doit être protégée des modules. Pour cela a été créée une Agence Liens qui encapsule les liens. Les services fournis par cette agence sont ceux définis dans le système OMPHALE pour la gestion des liens (détruire_lien, dupliquer_lien...).

Plus précisément, un module ne manipule ses liens que par l'intermédiaire de "*noms locaux*" qui sont des indices dans une table de liens appelée Environnement du module. Les primitives fournies par l'agence Lien rendent possible l'utilisation de ces noms locaux par l'encapsulation de l'Environnement.



Deux agences spécifiques OMPHALE

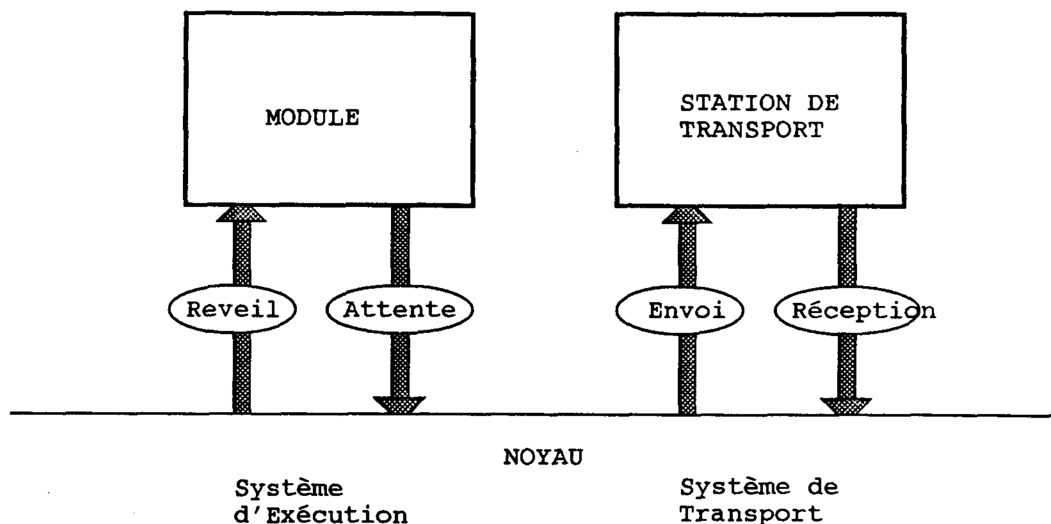
Les agences s'exécutent en mode système. Pendant l'exécution d'une primitive d'une agence, les structures internes sont accessibles. C'est de cette manière qu'est implanté le contrôle de l'utilisation des liens lors de l'envoi de messages.

3.1.5 Réalisation des modules OMPHALE

Pour chaque module, le noyau OMPHALE joue le rôle de :

1) **Système d'Exécution**:
C'est-à-dire organiser le séquencement attente–actif du module autour de la primitive ATTENDRE et des conditions de réveil.

2) **Système de Transport**:
C'est-à-dire réceptionner les messages provenant de la Station de Transport et les placer dans la Zone de Réception, et envoyer à la Station de Transport les messages de la Zone d'Emission.



Le rôle du noyau OMPHALE

Ce fonctionnement du noyau est basé sur les trois structures que sont les Zones d'Emission et de Réception et l'Environnement.

Il a été choisi d'implanter un module OMPHALE sous forme de deux tâches SPART.

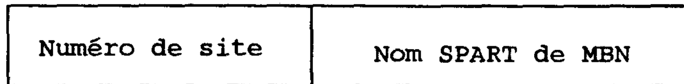
1) Une tâche TN qui va réaliser les actions noyau précédentes et cela en retrouvant dynamiquement les structures à utiliser dans le contexte du module. Cette tâche s'exécute en mode système.

2) Une tâche TC qui réalise en mode utilisateur la partie application du module.

Nous allons maintenant détailler la coopération de ces deux tâches.

A chaque tâche (TN et TC) est associée une boîte aux lettres SPART (MBN pour TN et MBC pour TC). Ces deux boîtes aux lettres permettent aux deux tâches de communiquer. MBN permet de plus la communication entre la tâche TN et la tâche TN correspondante de la Station de Transport du site.

Comme on le verra, c'est par l'intermédiaire des MBN que les modules envoient des messages OMPHALE, c'est donc naturellement à partir des noms systèmes SPART des boîtes aux lettres que s'effectuera le nommage OMPHALE. Un nom interne de module sera donc la concaténation d'un numéro de site et du nom système de sa boîte aux lettres MBN.

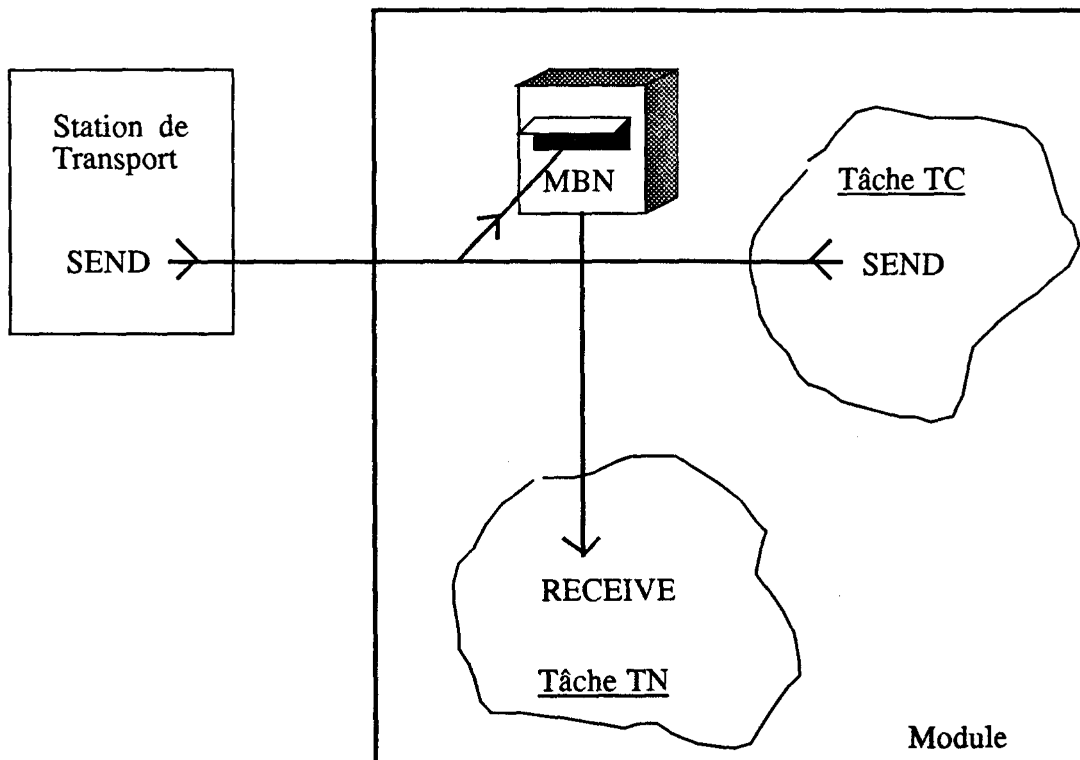


Nom interne de module

La boîte aux lettres MBN est utilisée dans les deux cas suivants:

1) L'arrivée d'un message OMPHALE à un module est signalée au TN du module par l'arrivée d'un message SPART en provenance de la Station de Transport. Le message contient un pointeur vers le message OMPHALE correspondant.

2) L'utilisation de la primitive ATTENDRE par le module est signalée à TN par l'arrivée d'un message SPART en provenance de TC. Le message contient un pointeur vers les conditions de réveil. Une zone mémoire partagée entre TN et TC joue le rôle de tampon pour la communication des conditions de réveil.



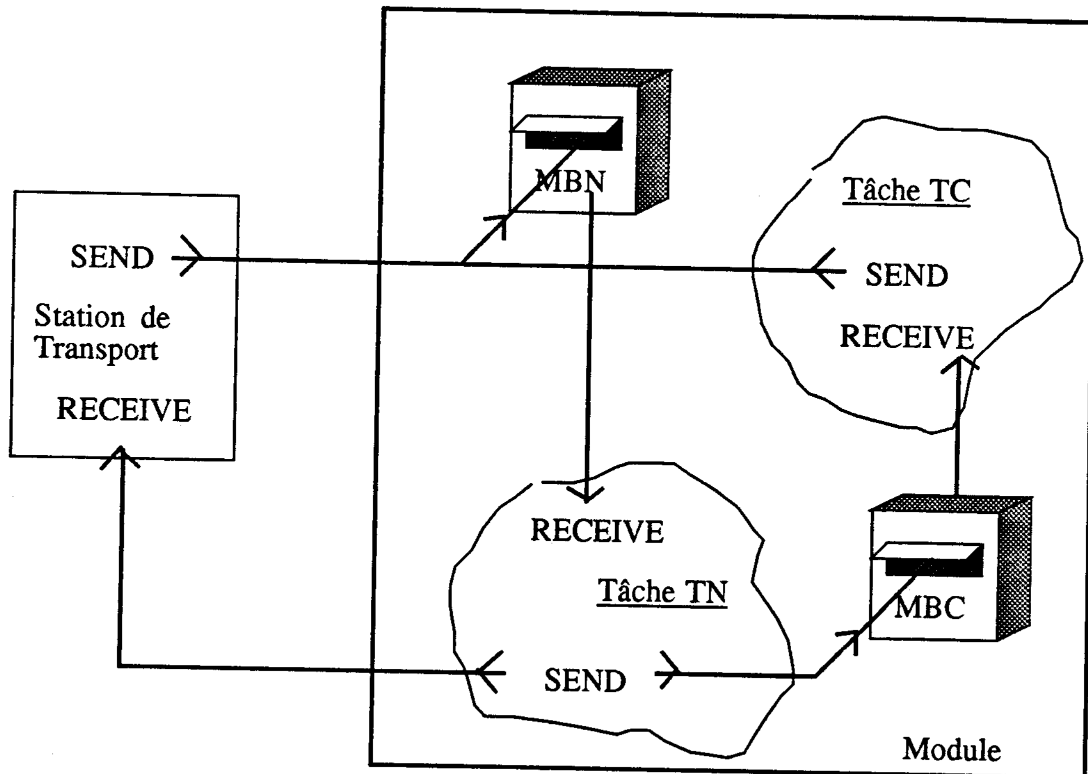
Le travail de TN est alors centré sur la réception de messages SPART sur la

boîte aux lettres MBN. Il peut être décrit par l'algorithme suivant :

```
Module TN :

algorithme principal:

    en_attente:=faux;
    boucle
      RECEIVE(message,MBN);
      si le message vient de TC alors
         traiter_mise_en_attente_de_TC(message);
      sinon  { le message vient de la station de transport }
         traiter_arrivee_message_omphale(message);
      fsi
    fin_boucle

traiter_mise_en_attente_de_TC(message_spart):

    en_attente:=vrai;
    recupération_des_conditions_de_reveil(message_spart);
    pour_chaque_message_omphale_de_la_zone_d_emission faire
       SEND(pt_vers_message_omphale,mbn_de_station_de transport);
    fait;
    si conditions_de_reveil_satisfaites alors
       SEND(pt_vers_message_a_traiter,MBC);
       en_attente:=faux;
    fsi

fin;

traiter_arrivee_message_omphale(message_spart):

    recuperer_pointeur_vers_message_omphale(message_spart);
    stocker_le_message_omphale_dans_zone_de_reception;
    si en_attente alors
       si conditions_de_reveil_satisfaites alors
          SEND(pt_vers_message_a_traiter,MBC);
          en_attente:=faux;
       fsi
    fsi

fin;
```

Du côté TC la réalisation de la primitive ATTENDRE se fait donc par un SEND sur

MBN des conditions de réveil suivi d'un RECEIVE sur MBC du message à traiter. On remarque que MBC ne contient au plus qu'un message indiquant quel est le message OMPHALE à traiter.



3.1.5.1 Station de Transport

L'envoi de messages (à la fin d'une étape de traitement) se fait par envoi de messages SPART à la Station de Transport du site. Un message SPART contient un pointeur vers un message OMPHALE.

La Station de Transport détermine grâce au champ destinataire des messages OMPHALE s'il y a lieu de réaliser le transfert vers un autre site ou non. Si oui, le message est transféré au site distant.

Toujours grâce au champ destinataire, la Station de Transport va déterminer quelle est la MBN du module destinataire et va y envoyer un message contenant un pointeur vers le message.

On remarque que les messages OMPHALE ne sont jamais recopiés d'un endroit à un autre de la mémoire d'un site. Les messages sont manipulés de façon interne aux TN par l'intermédiaire de pointeurs et de façon externe par l'intermédiaire des

noms systèmes provenant de l'agence Messages SPART. Tous les messages d'un site sont alloués dans un segment mémoire du site et c'est l'agence Messages qui gère complètement cet espace sous forme de multiples listes chaînées de messages.

Lorsqu'un transfert inter-sites est nécessaire, Unix voit le message OMPHALE à transférer par partage du segment messages. La coopération SPART-UNIX se fait par un mécanisme de boîtes aux lettres compatibles SPART et SYSTEM V. Le processus du coté UNIX va émettre le message grâce au mécanisme de "sockets" qui réalise un transfert fiable sur le réseau. A l'arrivée tout le mécanisme inverse est effectué (processus UNIX vers Station de Transport SPART).

3.1.6 Intérêts des Modules double-tâches

1) Le noyau SPART a permis l'implantation du noyau OMPHALE sous la forme, non pas d'un noyau unique prenant en charge toutes les structures de données associées à tous les modules, mais plutôt comme une tâche noyau associée à chaque module ne gérant que les structures du module. Cette méthode a la vertu de la simplicité. Il a été plus facile d'écrire cette tâche noyau plutôt qu'écrire un noyau unique. Cette méthode n'a pas augmenté la taille du code du noyau car la tâche noyau ayant le même code pour chaque module, celui-ci n'est présent qu'une fois en mémoire sous forme réentrante.

2) L'architecture du SPS7 est fortement modulaire et permet l'installation de plusieurs Modules de Traitement SPART. Avec deux Modules SPART il est possible, sans rien changer au système, de faire s'exécuter toutes les tâches TN sur un module et toutes les tâches TC sur l'autre module. Dans ce cas le travail global du système (somme du travail de tous les TN) ne pénalise en rien le travail purement lié aux applications (somme du travail de tous les TC). On aurait ici toute la puissance d'un 68010 pour exécuter les applications sans que le système ne consomme de puissance. Une telle architecture doit faire réfléchir finement à la répartition des tâches entre TN et TC. Eric Delattre et Jean-Marie Place [DEL86a] ont montré que l'optimum est d'avoir la même charge de travail du côté TN que du côté TC et donc que, dans nos conditions, l'optimum est un système à 100 % d'overhead !!!!

3.1.7 Récupération

Dans le système actuel, la récupération des modules isolés est réalisé par simple comptage des liens. Chaque module encapsule (dans TN) le nombre de liens vers lui-même dispersés dans le système.

Lorsqu'un module est créé, il possède un lien vers lui-même et le compteur de liens est mis à 1. Ensuite, à chaque duplication ou destruction de ce lien ce compteur va être incrémenté ou décrémenté. Cela se passe de la manière suivante :

Lorsqu'une agence Liens exécute une opération de duplication ou de destruction de liens, elle envoie un message SPART vers le module pointé par le lien (plus précisément vers la MBN de la tâche TN concernée).

L'algorithme précédent de TN est donc peu modifié : Une tâche TN est maintenant en attente de trois types de messages SPART : i) venant de TC ii) venant de la Station de Transport et iii) venant du noyau pour le comptage des liens.

Lorsqu'un tel compteur passe à la valeur du nombre de liens vers lui–même contenu dans l'environnement du module, le noyau entreprend la destruction du module lorsque celui–ci passe en attente.

3.1.8 Conclusion

La version ici décrite du noyau NERF d'OMPHALE fonctionne sur nos deux SPS7. La couche CORTEX est embryonnaire (Station de Transport, Serveurs de Noms, Serveurs de Modules existent dans des versions simples mais qui nous permettent de lancer et de suivre des applications réparties), une partie de cette couche est directement prise en charge par le noyau SPART (gestion fichiers, de la mémoire et de la périphérie).

Les développements du système OMPHALE s'orientent actuellement dans les directions suivantes:

1) Etude complète d'une couche CORTEX répartie (système de fichiers réparti, serveurs de noms hiérarchisés, interpréteur de commande, ...). L'enjeu est ici de valider le modèle utilisé pour la création d'une couche système. En particulier les problèmes de fiabilité doivent être réglés à ce niveau.

2) Création d'un langage de haut–niveau d'écriture de serveur autonomes.

D'autres aspects se devront d'être étudiés : Amélioration du noyau pour la récupération des modules isolés, régulation de charge et introduction de la migration de serveurs, mémoire secondaire de serveurs, tolérance aux pannes.

La plate–forme expérimentale SPS7 ne permet pas d'aborder concrètement les problèmes de performances: des mesures de performances mesureraient simplement celles du noyau SPART. Le développement de la couche CORTEX est le véritable enjeu de cette plate–forme, les réflexions de ce travail devraient nous amener au développement d'un système réel sur machine nue.

3.2 Architecture bi–processeur expérimentale

Parallèlement à l'expérimentation sur SPS7, il a été développé une architecture matérielle particulière dédiée au support du système OMPHALE. Le système OMPHALE tend au développement d'applications découpées en de nombreux processus communiquant par messages. Dans de telles conditions, le changement de processus est une opération fréquente, et cela entraîne sur une architecture classique un "*overhead*" système très important (c'est–à–dire que la machine passe beaucoup de temps à gérer les processus par rapport au temps passé par ceux–ci pour exécuter les applications). Le but de la réalisation de la machine expérimentale, appelée "**Site 0**", est d'expérimenter une architecture "*à changement de contexte rapide*" pour un tel système. La caractéristique importante de cette machine (qui n'est pas répartie actuellement, il ne s'agit que d'un site) est d'être formée de deux processeurs fonctionnellement différent : l'un exécute le noyau NERF et prépare les contextes d'exécution, l'autre exécute les serveurs OMPHALE. Un dispositif matériel spécifiquement développé permet la coopération des deux processeurs. Nous décrivons ici cette architecture et le support du noyau NERF.

3.2.1 Architecture matérielle du Site 0

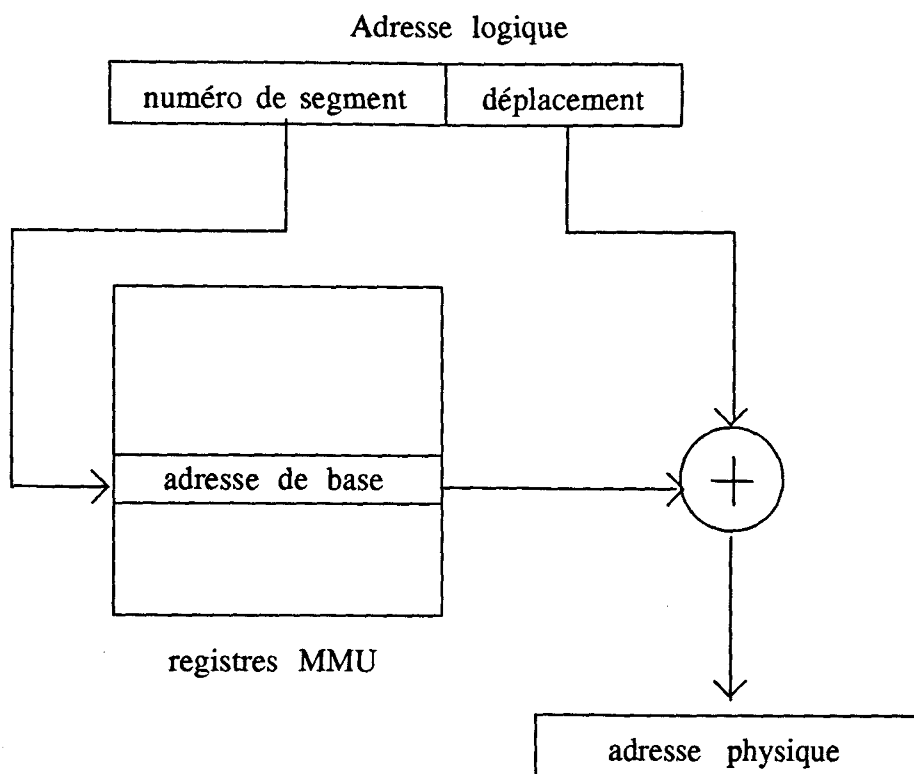
L'architecture matérielle du Site 0 comprend :

1) **Deux micro–processeurs Zilog Z8001**:
Le site 0 est donc une architecture bi–processeur. Les processeurs Z8001 sont des processeurs 16 bits à adressage segmenté. Chaque processeur est pourvu des équipements nécessaires à un fonctionnement autonome (mémoire privée RAM et ROM). L'un des processeur est appelé Processeur Noyau (PN), l'autre Processeur Calcul (PC).

2) **Un espace mémoire dit Mémoire de Modules**:
Cet espace doit servir à contenir les images des processus (modules) tournant sur la machine. Cette mémoire est découpée en blocs et chaque bloc est connecté à deux bus (BN et BC) utilisés l'un par PN, l'autre par PC pour accéder à la Mémoire de Modules.

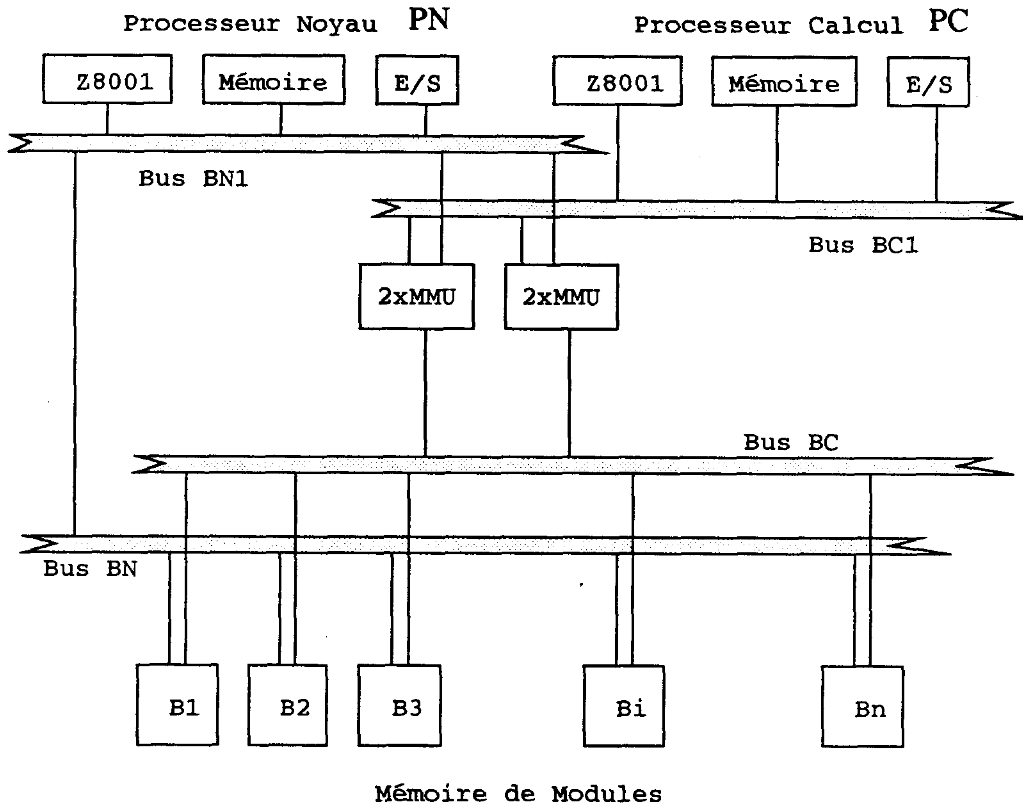
3) **Deux jeux de Memory Managment Unit (MMU)**:
Un boîtier MMU de Zilog associé à un Z8001 a le rôle de translater les adresses segmentées utilisées par le Z8001 en adresses physiques.
Une adresse logique est formée d'un numéro de segment et d'un déplacement dans le segment (limité à 64 koctets). Une MMU comprend 64 registres et chaque registre contient une adresse physique dite de base. La translation consiste à calculer une adresse physique en prenant celle contenue dans le registre de MMU

correspondant au numéro de segment de l'adresse logique et à y ajouter le déplacement.



Translation d'adresse

Le Site 0 comprend 2 jeux de deux boîtiers MMU. Ces deux jeux sont accessibles par chacun des deux processeurs à partir de leur bus (appelés BN1 et BC1). En sortie, les MMU sont toutes connectées au Bus BC.

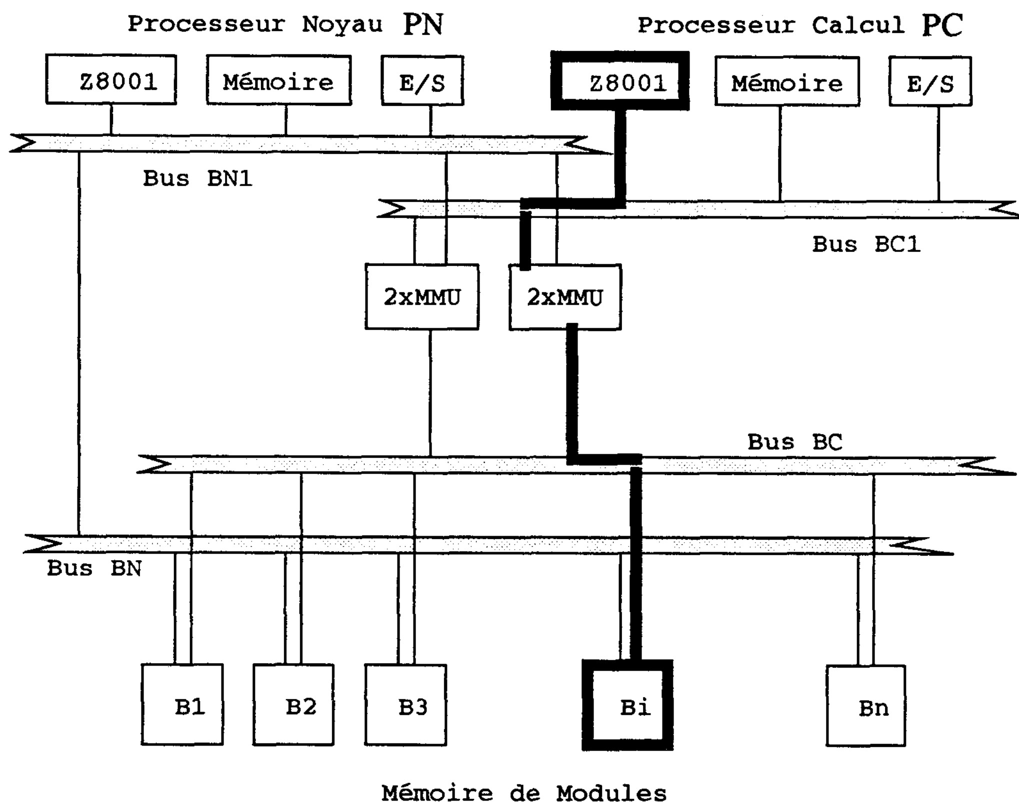


Architecture du Site 0

Le partage de la Mémoire de Modules entre les deux processeurs se fait par l'utilisation des jeux de MMU. Nous détaillons ce mécanisme dans la section suivante.

3.2.2 Partage de la Mémoire de Modules

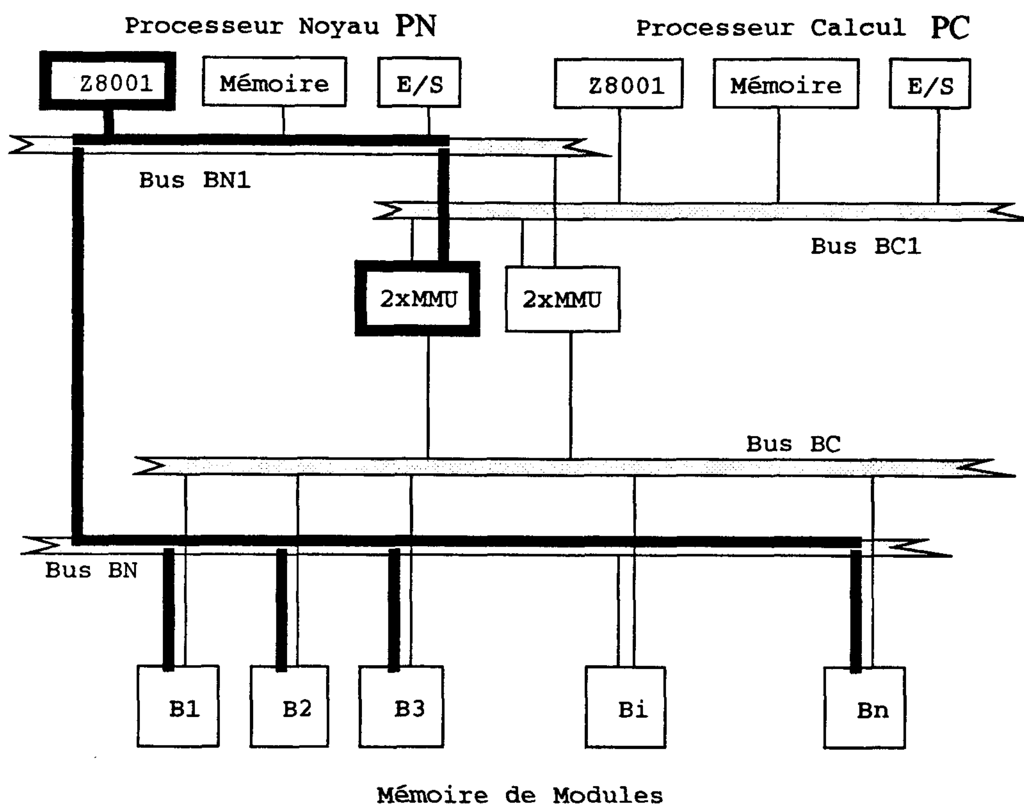
A chaque instant, le processeur PC est connecté à un bloc de la Mémoire de Modules par l'intermédiaire d'un jeu de MMU. Autrement dit, les registres de la MMU utilisée par PC contiennent des adresses physiques dans un seul bloc de la Mémoire de Modules.



Chemin d'accés du processeur Calcul

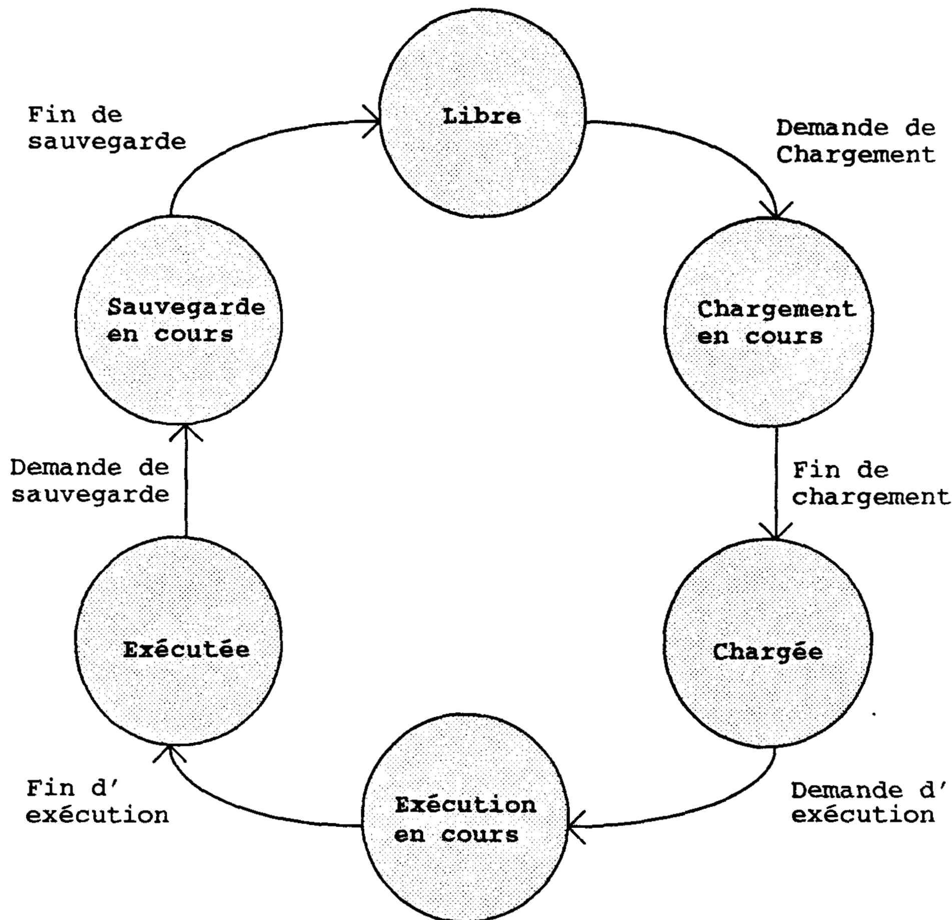
Pendant que PC utilise le bloc Bi, le Processeur Noyau peut accéder directement (sans passer par les MMU) à la Mémoire de Modules. Il ne doit pas accéder au bloc utilisé par PC, cela est ici simplement assuré par certification des logiciels tournant sur PN.

Le processeur Noyau a aussi accès au jeu de MMU non utilisé par PC, non pas pour la translation d'adresses mais pour chargement et sauvegarde des registres.



Chemin d'accés du processeur Noyau

Le partage des circuits MMU entre les deux processeurs se fait par une circuiterie spécifique [DEL85b] qui contrôle le "**graphe d'état**" de chaque MMU. Cette circuiterie a été développée par l'équipe OMPHALE.



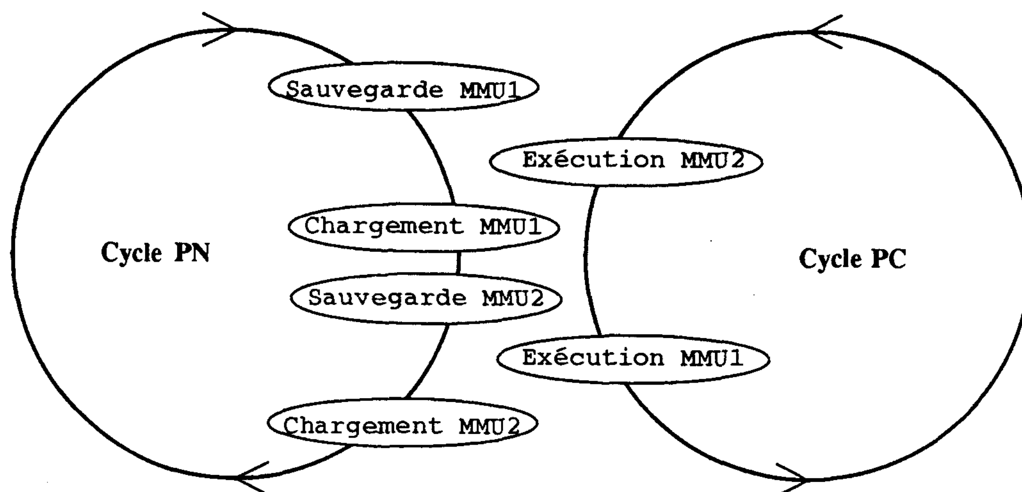
Graphe d'état d'une MMU

Le partage des MMU se fait de la manière suivante et se fonde sur le partage des tâches entre les deux processeurs : PN réalise les chargements et sauvegardes des MMU et PC utilise les MMU pour travailler sur le bloc qui lui est connecté.

Donc PN effectue sur la circuiterie spécifique les Demandes de Chargement, les Fins de Chargement, les Demandes de Sauvegarde et les Fins de Sauvegarde, tandis que PC effectue les Demandes et les Fins d'Exécution.

Lorsqu'une Demande ne peut être satisfaite (aucune MMU n'est dans le bon état), le processeur est dérouté vers une attente active de réessais.

Lorsqu'une Demande peut être satisfaite la circuiterie connecte le processeur à la MMU qui se trouve dans le bon état.



Travaux parallèles de PN et PC

Comme on le voit, l'exécution se réalise donc en parallèle avec le chargement et la sauvegarde. Cela est à comparer avec le cycle d'un processeur unique qui enchaîne séquentiellement les opérations de Chargement–Exécution–Sauvegarde.

3.2.3 Processus et Changements de contexte

Un processus est défini sur une machine à adressage segmenté par l'ensemble des segments qui définissent son image mémoire (segment de code , segment de pile, segments de données), c'est à dire par l'ensemble des valeurs des registres de translation d'adresse des MMU auxquels il faut ajouter le vecteur d'état du processeur.

Sur le Site 0, l'image mémoire d'un processus et l'espace de stockage des registres MMU va toujours se trouver entièrement dans un bloc de la Mémoire de Modules. Un bloc peut bien sûr contenir plusieurs de ces images mémoires.

Pour un système multi–processus, le **Changement de contexte** consiste à faire changer le processus en cours d'exécution (soit parce que le processus a épuisé la tranche de temps qui lui était allouée, soit parce qu'il passe en attente d'un événement extérieur).

Sur une machine classique, le changement de contexte consiste à sauvegarder le

contenu des registres des MMU et le vecteur d'état, puis charger la MMU et le registre d'état avec les nouvelles valeurs pour le nouveau processus. Sur le Site 0, qui possède deux jeux de MMU ces opérations sont effectuées par PN et donc ne coûte rien au processeur de calcul PC. **Le processeur PC est donc un processeur à changement de contexte instantané !!!.**

Pour que cela soit effectivement le cas, il faut néanmoins que les deux conditions suivantes soient réalisées:

1) Le processus suivant, lors d'un changement de contexte, ne doit pas avoir son image mémoire dans le même bloc que le processus précédent . En effet dans ce cas PN ne peut anticiper le chargement car il n'a pas accès à ce bloc pendant l'exécution du processus précédent.

2) Le processus suivant lors d'un changement de contexte ne doit pas être "calculé" par le processus en cours. Dans ce cas PN ne peut pas faire de préchargement d'un processus qu'il ne connaît pas encore. C'est souvent le cas lors qu'un processus en réveille un autre en se mettant en attente. Le processus suivant dépend du processus précédent.

Pour que ces conditions soient réalisées, il est important que le nombre de modules prêts à être lancés à chaque instant soit assez grand vis-à-vis du nombre de processus existants. Dans ce cas, pendant l'exécution d'un processus il est toujours possible d'en trouver un à précharger dans un bloc différent du bloc connecté à PC et les dépendances entre processus vont se trouver entrelacées et donc ne pénaliseront pas les possibilités de préchargement. Le modèle Serveur vérifie cette propriété grâce à ses nombreux processus communicants et à sa communication asynchrone qui rend indépendants les modules prêts.

3.2.4 Implantation de NERF

Le principe de l'implantation est de faire exécuter par PC les phases de traitement des modules OMPHALE et de laisser à la charge de PN l'exécution de NERF (par analogie avec l'implantation sur SPS7, PN exécute l'équivalent des tâches SPART TN, et PC les tâches SPART TC).

3.2.4.1 Structures de Données

Les trois structures de données (Zone d'Emission, Zone de Réception, Environnement) caractérisant un module sont implantées de la manière suivante:

1) Les Zones de Réception des modules sont implantées du côté PN. Seul NERF doit les manipuler. Elles sont chaînées dans un tampon.

2) Chaque processus sur PC voit sa Zone d'Emission comme un segment dans le

bloc qui lui est connecté. Les primitives de manipulation de cette Zone s'exécutent en mode système du côté PC (le code de ces primitives est dans la mémoire privée de PC).

3) L'environnement est aussi implanté comme un segment dans le bloc associé au processus de PC. Il est accédé en mode système par PC pour valider les opérations sur la Zone d'Emission. Il est accédé par PN lors de l'opération de réveil du processus, mais, à ce moment, PC n'est pas connecté à ce bloc.

3.2.4.2 Schéma d'exécution

Détaillons le cycle d'un module OMPHALE :

Attente sélective–Réception–Traitement–Envoi

1) **Attente sélective**:
Grâce à la Zone de Réception, PN a toutes les informations pour décider du réveil du module. Lorsque c'est le cas, PN choisit un message à traiter et le place dans le segment pile du processus. Il doit préparer le message (traduction des noms locaux) en accédant à l'Environnement dans le bloc contenant l'image mémoire. Lorsque ce travail est fait, PN place le processus dans une file d'attente des processus prêts.

2) **Réception**:
PN fait une demande d'acquisition d'une MMU pour chargement. La réponse peut être négative, dans ce cas PN s'occupe d'un autre module. En cas d'obtention d'une MMU, PN charge les registres de la MMU de manière à limiter l'espace d'adressage de PC à l'image mémoire du processus à lancer. PN termine par une Fin de Chargement. PC commence la phase de traitement.

3) **Traitement**:
La phase de traitement est exécutée par PC. Le processus retrouve le message à traiter sur sa pile. La fin de la phase de traitement consiste à libérer la MMU (Fin d'Exécution). Le processeur PC peut immédiatement demander une nouvelle MMU (Demande d'Exécution) pour exécuter une phase de traitement pour un autre module.

4) **Envoi**:
Lorsque PN fait une Demande de Sauvegarde, il va acquérir une MMU d'un processus qui vient de s'arrêter. Il va alors sauvegarder les registres de la MMU puis la libérer (Fin de Sauvegarde). Il va aussi récupérer la Zone d'Emission du processus et déposer les messages dans les Zones de Réception correspondantes. Il doit aussi stocker les conditions de réveil dans la Zone de Réception du processus.

On peut considérer le travail de PN comme:

1) **Un travail de préparation**:
Il s'agit, pour un module donné, de i) distribuer les messages émis ii) réaliser l'attente sélective, ii) préparer le message à traiter.

2) **Un travail de chargement**:
Il s'agit de charger le jeu de MMU pour un module prêt.

2) **Un travail de sauvegarde**:
Il s'agit de sauvegarder le jeu de MMU pour un module en attente.

Le travail de PC est un **travail d'exécution** des phases de traitement.

Une simulation de ces travaux par les deux processeurs PN et PC a été réalisée [DEL86a]. Un des paramètres est de savoir quel travail (Préparation, Chargement, Sauvegarde) PN doit choisir de faire en priorité. La réponse de la simulation a été de privilégier soit le Chargement soit la Sauvegarde au détriment de la Préparation.

La Simulation a aussi montré que, pour ces priorités, il n'est pas nécessaire d'installer plus de deux jeux de MMU.

De plus, la simulation montre que les performances de la machine sont optimales lorsque le temps d'exécution des modules est égal aux temps cumulés de Préparation, Chargement, Sauvegarde. Ce résultat montre que l'architecture est bien adaptée aux systèmes dans lequel les processus sont nombreux et enchaînent des phases de traitement assez courtes. La notion de processus Serveur tend à un système ayant cette propriété.

3.2.4.3 Communication Asynchrone

Le Schéma de communication du système OMPHALE repose sur le fonctionnement suivant :

Un module exécute une phase de traitement, puis s'arrête en envoyant des messages. Un certain nombre de ces messages rendent prêts les modules destinataires. Ces modules vont ensuite être réveillés pour exécuter une phase de traitement, etc...

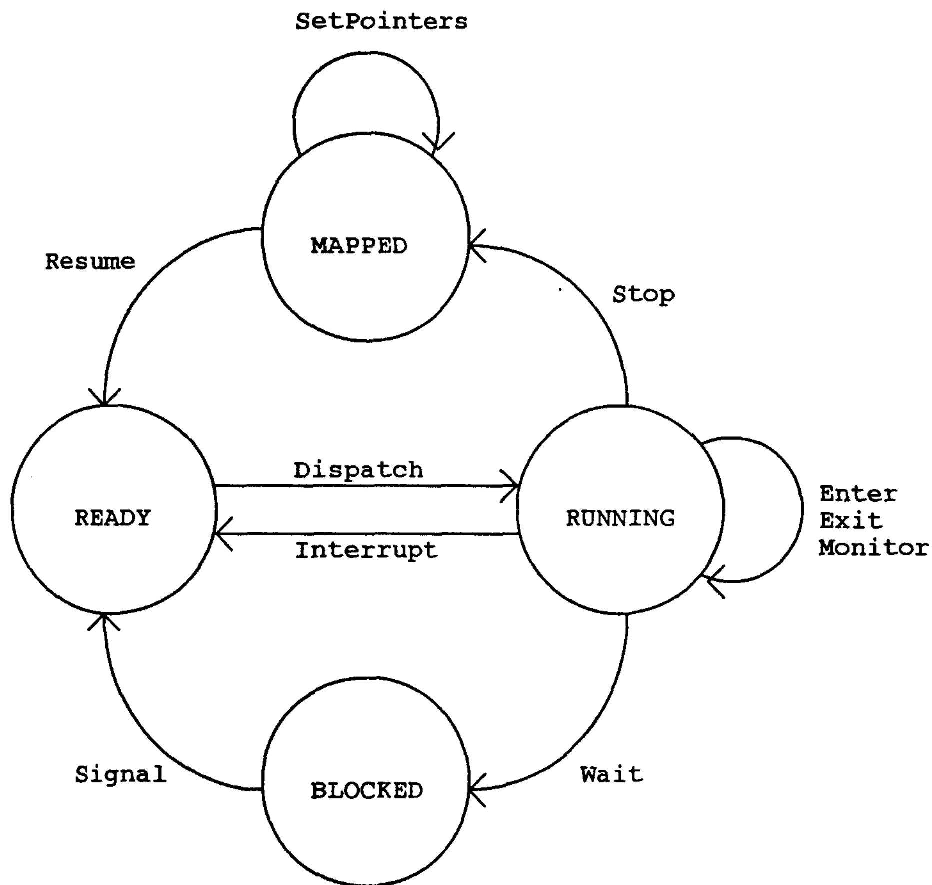
Ce fonctionnement simple n'entraîne pas une dépendance temporelle forte des exécutions des modules prêts. L'ordre d'exécution des modules prêts peut être choisi en fonction de considérations systèmes telle l'allocation des modules dans les blocs. Il faut néanmoins se prémunir des problèmes de "*famine*", c'est-à-dire éviter qu'un module prêt n'obtienne jamais le processeur.

3.2.4.4 Ecriture de NERF

L'écriture du noyau NERF a été précédée de l'écriture d'un noyau de multi-programmation baptisé EXO sur le site 0 . Ce noyau a été écrit en MODULA 2

[WIR83] à l'aide d'un cross–compilateur tournant sur le système MULTICS [ORG72] d'un DPS8 de BULL. Ce noyau implante des processus primitifs et leurs synchronisations par *"moniteurs de Hoare"*. Les moniteurs utilisés sont analogues à ceux du langage MESA [LAM80]: c'est à dire que l'opération "signal" ne fait pas perdre le processeur à celui qui l'exécute. Cette méthode permet d'éviter de trop fréquents changements de contexte lors de l'utilisation des moniteurs.

Le graphe de transition d'un processus primitif est le suivant :

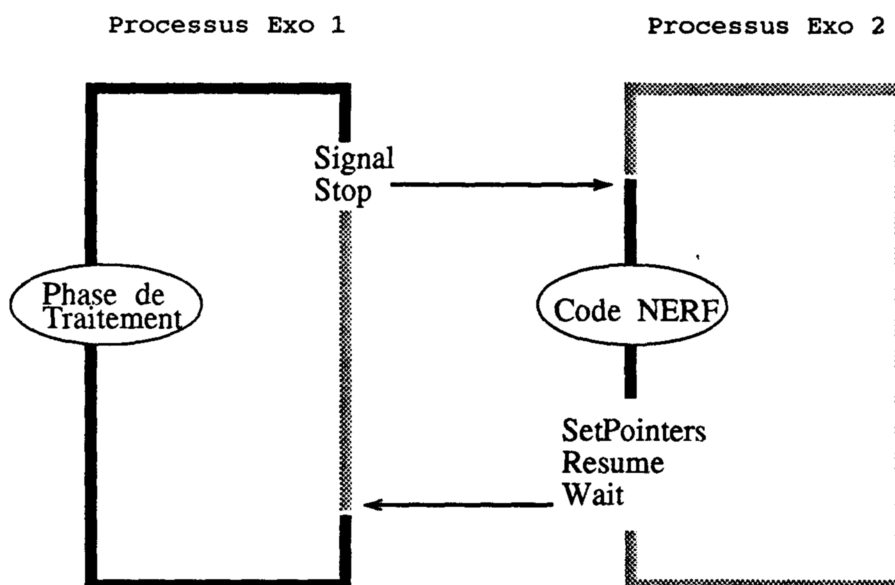


Graphe de transition d'un processus primitif

Les processus EXO sont des processus *"enveloppes"* [HOL83]: L'opération SetPointers permet de positionner les registres internes du processeur et donc de changer l'exécution du processus vers un autre code. Par ce mécanisme, c'est un processus EXO qui va prendre en charge les phases de traitement à exécuter.

Sur PC, NERF a donc été réalisé comme deux processus primitifs, l'un prend en charge par le mécanisme précédent les phases de traitement des modules, l'autre exécute le code NERF de PC qui enchaîne ces phases de traitement.

Ces deux processus primitifs se synchronisent par un moniteur: La fin d'une étape de traitement est réalisée par un "**Signal**" sur une condition du moniteur suivi d'un "**Stop**". Le signal réveille le processus primitif NERF, celui-ci va faire un "**SetPointers**" sur l'autre processus pour que celui-ci exécute ensuite la phase de traitement prévue, un "**Resume**" pour le libérer, puis se mettre en attente par "**Wait**".



Synchronisation des 2 processus primitifs de NERF

3.3 Conclusion

La simplicité du modèle Serveur qui entraîne une faible interaction entre le noyau qui le supporte et les modules serveurs, permet ici d'aborder la parallélisation matérielle du système OMPHALE (Un travail de parallélisation du système CHORUS dans sa version avec étapes de traitement existe aussi mais d'une façon différente de la nôtre, voir [LEB88]).

Le développement du site 0 montre que la prise en charge de nombreux processus

peut se concevoir sur des architectures matérielles non classiques à condition que le modèle d'exécution ait de bonnes propriétés. Un modèle comme le modèle Serveur qui préconise des entités très autonomes, ayant un fonctionnement de type automate, et des points de coopération très simples (une seule primitive d'envoi et de réception) a ici des bonnes propriétés. Notre machine peut servir à la gestion de très nombreux processus tout en gardant au système supporté des performances acceptables.

Chapitre 4. Conclusion

A partir du Modèle Objet nous avons défini un modèle d'objets actifs dit **Modèle Serveur** permettant la prise en compte du parallélisme entre objets ainsi que la distribution des objets sur un ensemble de sites.

Ce Modèle se fonde sur l'équation **un objet = un processus** et organise la coopération des objets par l'envoi de **requêtes de service** au niveau inter–processus étendu à l'ensemble des sites. L'étude de ce modèle fait apparaître la notion de **Contrôleur**: Chaque objet doit être muni d'un **contrôleur** qui règle les arrivées asynchrones de requêtes de service.

Le modèle proposé se veut simple et uniforme:

- Un seul type d'entité: Le Serveur.
- Un seul mode de coopération: La requête de service.
- Un seul comportement: Le traitement séquentiel mais ordonné par le contrôleur des requêtes reçus (fonctionnement en automate).

La notion de Serveur s'inscrit très bien dans la problématique des systèmes répartis structurés en termes de processus communiquants par messages. Face aux choix de conception, généralement ouverts dans le panorama des systèmes existants, le Modèle Serveur apporte des éléments de réponse:

- Un objet par contexte d'exécution.
- La communication asynchrone (pas d'appel de procédure à distance mais simple transfert de messages).
- L'absence de la notion de portes (abandon du modèle producteur consommateur).
- Pas de gestion des gros transferts (données encapsulées).

Les aspects classiques du nommage dans les systèmes répartis et en particulier la transparence de la répartition permettent la distribution des serveurs sur un ensemble de sites. Cela permet une exploitation réelle du parallélisme du modèle.

La protection nécessaire dans un système se réalise bien par l'utilisation de

Capacités qui contrôlent à la fois la désignation et l'utilisation des services.

Notre expérience dans la définition du modèle Serveur et dans les réalisations expérimentales nous conforte dans l'idée initiale que le modèle objet étendu au parallélisme et à la distribution peut être choisi comme modèle d'exécutif réparti. Et cela bien que certain aspects tels la tolérance aux pannes et la régulation de charge (par migration) n'ait pas été étudiés dans notre étude.

Le noyau NERF est le résultat de l'utilisation du Modèle Serveur comme base d'un exécutif réparti. NERF supporte la création, la coopération et la destruction de serveurs sur un ensemble de site. Pour définir NERF à partir du Modèle Serveur, nous avons:

- implanté la notion de requête de service par l'utilisation d'un mécanisme de transfert asynchrone de message uniforme sur l'ensemble des sites. L'espace de stockage des messages en cours de transfert est découpé en deux espaces distincts: Un appartenant au contexte de l'émetteur du message, l'autre dans le contexte du récepteur. Cela offre les avantages, pour le premier, de pouvoir réaliser un contrôle global des messages avant transfert, et pour le deuxième de pouvoir réaliser facilement un mécanisme d'attente sélective de messages.

- implanté la notion de contrôleur par l'utilisation d'une primitive unique d'envoi–attente–réception . Un serveur paramètre localement cette primitive (en validant des services), il s'impose ainsi un fonctionnement en automate qui assure la cohérence de son état.

- fournit les possibilités de définition de serveurs autonomes réutilisables par un mécanisme de nommage permettant le transfert de noms de serveurs dans les messages. L'utilisation de noms locaux et de liens permet de plus de fournir un domaine de protection au niveau du service.

L'ensemble des ces mécanismes permettent la réalisation d'un noyau conservant simplicité et uniformité. Etant à la fois support du modèle Serveur et d'exécutions réparties, NERF est une base qui nous semble prometteuse, à la fois pour la prise en charge des Langages Orientés Objets Parallèles et pour la création de Systèmes d'Exploitation Répartis d'Objets., deux enjeux qui semblent clairement être d'avenir.

La validation complète de notre modèle ne pourra se faire que lors de la construction de la couche CORTEX du système. Ces travaux sont en cours actuellement sur la base d'une plate–forme expérimentale NERF opérationnelle sur nos deux SPS7. Ces travaux pourraient entraîner des modifications des spécifications du noyau, nous pensons qu'elles seront peu nombreuses et ne changeront pas profondément le modèle présenté ici. Le développement de la couche CORTEX est le véritable enjeu de cette plate–forme, les réflexions de ce travail devraient nous amener au développement d'un système réel sur machine

nue.

Les performances à partir du Modèle Serveur semblent pénalisées par les fréquents changements de contexte issus de la coopération de nombreux processus. Un élément de réponse est la plate-forme matérielle conçue spécifiquement pour NERF: La simplicité du modèle Serveur qui entraîne une faible interaction entre le noyau qui le supporte et les modules serveurs, permet ici d'aborder la parallélisation matérielle du système OMPHALE. Un dispositif matériel cache ici les temps de changement de contexte derrière le parallélisme de la machine. Le développement de ce "Site 0" montre que la prise en charge de nombreux processus peut se concevoir sur des architectures matérielles non classiques à condition que le modèle d'exécution ait de bonnes propriétés. Le modèle Serveur qui préconise des entités très autonomes, ayant un fonctionnement de type automate, et des points de coopération très simples (une seule primitive d'envoi et de réception) a ici ces bonnes propriétés.

Pour l'avenir, si le modèle Serveur s'avère séduisant de par sa simplicité et son uniformité, un certain nombre de questions restent ouvertes, et devront être étudiées pour permettre la construction réelle d'un système complet. On peut citer certaines catégories de questions:

- Toutes celles concernant la réalisation des grandes fonctionnalités d'un système d'exploitation réparti: système de gestion de fichiers, chaîne de fabrication des modules, gestion des utilisateurs... Il paraît inéluctable que l'approche de conception par objets actifs modifiera radicalement la construction de ces fonctionnalités. L'enjeu est aussi ici de montrer que les mécanismes de base du modéle ne sont pas contradictoires avec ces objectifs, en particulier la gestion particulière des capacités et l'absence de mécanismes particuliers pour la transfert de gros messages seront lourds de conséquences.

- Les questions concernant la migration. Ce point n'a pas été abordé dans ce travail. Néanmoins, les mécanismes de nommage utilisés, ainsi que l'utilisation d'une Station de Transport des messages en dehors du noyau sont une base qui doit permettre l'intégration de la migration des processus serveurs sans remettre en cause le modèle.

- Les aspects de tolérance aux pannes. Là encore, le modèle serveur offre des possibilités intéressantes. Le fonctionnement en automate des serveurs pourrait être amélioré pour rendre atomique l'exécution des services et l'envoi final des messages. Cela passe par l'étude du traitement des exceptions et l'utilisation de serveurs couplés (par référence aux acteurs couplés de Chorus). Une telle approche conserverait la simplicité et uniformité du modèle.

- La prise en charge des Langages Orientés Objet (parallèles ou non). Les aspects méthodologiques des Langages Orientés Objet, telle la hiérarchie de classes (avec les liens d'héritage ou de délégation), sont des aspects toujours fortement centralisés. L'introduction de la répartition (et du parallélisme) est ici neuve. Une reflexion profonde est ici nécessaire pour déboucher sur des langages de haut niveau adaptés au modéle serveur.

- Finalement les questions de performances. NERF peut-il d'adapter aux stratégies et mécanismes permettant de les améliorer (caches d'accélération, mémoire virtuelle étendu au réseau, architecture matérielle des sites fortement parallèle...)? La simplicité du modèle est ici positive (elle nous permet dans le Site 0 de distribuer matériellement l'activité d'un site) , mais peut s'avérer insuffisante dans d'autres cas.

Il reste cependant légitime de considérer
que l'un des énoncés fondamentaux de la théorie de l'information,
à savoir que la transmission d'un message s'accompagne nécessairement
d'une certaine dissipation de l'information qu'il contient,
est l'équivalent en informatique du deuxième principe en thermodynamique.

Jacques Monod.
"*Le hasard et la nécessité*"

Chapitre 5. Bibliographie

[ACC86] M. Accetta & all.
Mach: A New Kernel Foundation for UNIX Development
Proc. Summer 1986 USENIX Technical Conference and Exhibition, (june 1986) pp. 93,112

[AGH87] G. Agha, C. Hewitt
Concurrent Programming Using Actors
in Object–Oriented Concurrent Programming, MIT Press (1987), pp.37,54

[ALM85] G.T. Almes, A.P. Black, E.D. Lazowska, J.D. Noe
The EDEN system: a technical review
IEEE trans. on Software Engineering, vol. SE–11, 1 (January 1985), pp. 1214,1224

[BAL86] R.Balter & all
Principes de conception du système d'exploitation réparti GUIDE
BIGRE+GLOBULE numéro 52 (décembre 1986) pp. 3,23

[BAN80] J.S. Banino & all
CHORUS: an Architecture for Distributed Computing
3rd Internatiaonal Conference on Distributed Computing Systems, Ft Lauderdale, Rapport INRIA (nov 1980)

[BAS77] F. Baskett, J.H. Howard, J.T. Montague
Task Communications in DEMOS
Proc. of th 6th Symposium on Operating Systems Principles (November 1977), pp. 23,31

[BIR85] A.D. Birrell
Implementing Remote Procedure Calls
ACM Trans. on Computer Systems, vol. 3, 1 (February 1985), pp 1,14

[BLA85] A.P. Black
Supporting distributed applications: experience with EDEN
Proc. 10th ACM Symp. on Operating Systems Principles, SIGOPS Operating Systems Review, vol 19, 5 (December 1985), pp. 181,193

[BLA86] A.P. Black, N. Hutchinson, E. Jul, H. Levy
Objet structure in the Emerald system
Proc. First ACM Conf. on Object Oriented Systems, Languages and Applications (OOPSLA), Portland (September 1986)

[BLA87] A.P. Black, N. Hutchinson, E. Jul, H. Levy, L. Carter
Distribution and Abstract Types in Emerald
IEEE Transactions on Software Engineering, Vol. 13,1 (Jan 1987) pp. 65,76

[BOO83] G. Booch
Software Engineering with ADA
Reading, Mass: Addison–Wesley (1983)

[BRO84] F. Browaeys, H. Derriennic, P. Desclaud, H. Fallour, C.Faulle, J. Febvre, J.E. Hanne, M. Kronental, J.J. Simon, D. Vojnovic
Sceptre: proposition de noyau normalise pour les exécutifs temps réel
Technique et Science Informatique, vol. 1 (Novembre 1984) pp. 45,62

[BUL85a] BULL (Cie)
Structure Générale du SPS7: Manuel de présentation
(Janvier 1985)

[BUL85b] BULL (Cie)
Système d'exploitation SPART: Manuel de présentation
(Octobre 1985)

[CHE82] D.R. Cheriton
The Thoth System: Multi–process Structuring and portability
Elsiver, New–York (1982)

[CHE83] D.R. Cheriton, W. ZWaenepoel
The distributed V kernel and its performance for diskless workstations Proc. on Operating Systems

Principles, ACM, New–York (1983), pp. 128,140

[CHE84] D.R. Cheriton
The V kernel, a software base for distributed systems
IEEE Software, vol. 1, 2 (1984), pp. 19,42

[CHE84b] D.R. Cheriton
An experiment using registers for fast message–based interprocess communication
Operating Systems Review, vol. 18 (October 84), pp. 12,20

[CHE85] D.R. Cheriton, W. Zwaenepoel
Distributed Process Groups in the V Kernel
ACM Trans. on Computer Systems, vol. 3,2 (May 1985), pp. 77,107

[CHE88] D.R. Cheriton
The V Distributed System
Comm. ACM, vol. 31,3 (March 1988) pp. 314,333

[COR81] CORNAFION
Systèmes informatiques répartis, concepts et techniques
Dunod (1981)
Distributed Computer Systems: Communication, Cooperation, Consistency
Elsevier (1985)

[COX86] B.J. Cox
Object Oriented Programming: An Evolutionary Approach
Addison–Wesley (1986)

[DEL84] E. Delattre, J.M. Geib
A Distributed and Protected System, which Maintains Modularity Up to Execution
Proc. Conf. on Hardware supported implementation of concurrent languages in distributed systems, Bristol (mars 1984)

[DEL85a] E. Delattre, J.M. Geib, J.F. Mehaut
The Omphale Distributed System Architecture: Communication and Concurrency Issues
Parallel Computing 85, Berlin (Septembre 1985)

[DEL85b] E. Delattre, J.M. Geib, J.M. Place *Distributed*

Hardware for V.L.S.I. Component Sharing
Proc. Euromicro 85 , Bruxelles (septembre 1985)

[DEL86a] E. Delattre, J.M. Place
Le support système des méthodologies de programmation orientée objet: 100 % d' "overhead"!
Journées AFCET GROPLAN (janvier 1986)

[DEL86b] E. Delattre, J.M. Geib, J.F. Mehaut, J.M. Place
Two Implementations of the OMPHALE Distributed Architecture
IASTED International Symposium Applied Informatics (Janvier 1988)

[DEN66] J.B. Dennis, E.C. Van Horn
Programming Semantics for Multiprogrammed Computations
Comm. ACM, vol. 9,3 (March 1966)

[DEN76] P.J. Denning
Fault Tolerant Operating Systems
Computing Surveys, vol. 8,4 (December 1976)

[DOD84] U.S Department of Defense
ADA Reference Manual

[ESP86] Rapport ESPRIT *Portable Common Tool Environment, Functionnal Specifications*
(1985)

[FAB74] R.S. Fabry
Capability-based addressing
Comm, ACM, vol. 17,7 (July 1974) pp. 403,412

[FIN81] U. Finger, G. Médigue
Architectures multi-microprocesseurs et disponobilité: la SM90
L'écho des recherches n0 105, (Juillet 1981), pp. 15,21

[FIT86] R. Fitzgerald, R.F. Rashid
The integration on virtual memory management and interprocess communication in ACCENT
ACM Trans. on Computer Systems, vol. 4, 2 (May 1986), pp. 147,177

[GEI86] J.M. Geib
OMPHALE: système réparti orienté objet 3èmes

journées d'études de l'AFCET sur les Langages Orientés Objet, Paris (janvier 1986)

[GOL83] A. Goldberg, D. Robson, D. Ingalls
Smalltalk-80: The Language and its Implementation
Addison-Wesley (1983)

[GOL84] A. Goldberg
Smalltalk-80: The Interactive Programming Environment
Addison-Wesley (1983)

[GUI82] M. Guillemont
The CHORUS distributed operating system: design and implementation
International Symposium on Local Computer Networks, Florence, Italy, (April 1982), pp. 207,223

[GUI84] M. Guillemont, H. Zimmermann, G. Morisset, J.S. Banino
CHORUS: Une architecture pour les systèmes répartis
Rapport de recherche INRIA 274 (mars 1984)

[HAN78] P. Brinch-Hansen
Distributed Processes: A Concurrent Programming Concept
Comm. ACM, vol. 21,11 (November 1978) pp. 934,941

[HAN81] P. Brinch-Hansen
Edison. A multi-processor language
Software-Practice and experience, vol. 11,4 (April 1981), pp. 325,361

[HER87] F.Herrmann
CHORUS: Un Environnement pour le développement et l'exécution d'applications réparties
Technique et Science Informatique, vol. 6,2 (mars 1987) pp. 162-165

[HEW77] C. Hewitt
Viewing Control Structures as Patterns of Passing Messages
A.I. Journal, vol 8,3 (june 1977) pp. 323,364

[HOA74] C.A.R. Hoare
Monitors: an operaring system structuring concept
Comm. ACM, vol. 18,2 (February 1975), pp. 95

[HOA78] C.A.R. Hoare
Communicating sequential processes
Comm. ACM vol. 21,8 (august 1978) pp. 10,23

[HOL83] R.C. Holt
Concurrent Euclid, the Unix system and Tunis
Addison–Wesley, Reading, Massachusettse (1983)

[HUT85] D.Hutchinson, J. Mariani, D. Shepherd
Local area networks: an advanced course
Lecture Notes in Computer Science no 184, Springer–Verlag (1985)

[INM84a] Inmos (Cie)
OCCAM Programming Manual
Prentice–Hall International (1984)

[JON79] A.K. Jones
The object model: a conceptual tool for structuring software
in R. Bayer, R.M. Graham et G.Seegmuller, éditors, Operating Systems, an Advanced Course, Springer–Verlag, New–York, U.S.A. (1979)

[JON86] M.B. Jones, R.F. Rashid
Mach and Matchmaker: kernel and langage support for object oriented distributed systems
Proc. First ACM Conf. on Object Orientd Systems, Languages and Applications (OOPSLA), Portland (September 1986), pp. 67,77

[KER84] B.W. Kernighan, R. Pike
The UNIX Programming Environment
Prentice–Hall (1984)

[KIE78] R.B. Kieburtz, A. Silberschatz
Capability managers
IEEE Trans. Software Eng., vol. SE–4 (November 1978), pp. 467,477

[KRA87] S. Krakowiak
Systèmes d'exploitation répartis: progrès récents et tendances de la recherche
Technique et Science Informatiques, vol. 6 (mars 1987)

[LAM74] B.W. Lampson *Protection*

Proc. 5th Princeton Symposium on Information Science and Systems, Princeton University (1971) pp. 437,443. Operating Systems Review, vol. 8,1 (1974) pp. 18,24

[LAM80] B.W. Lampson, D.D. Redell
Experience with processes and monitors in Mesa
Comm. ACM, vol. 23,.2 (1980), pp. 105,117

[LAZ81] E.D. Lazowska, H.M. Levy, G.T. Almes, M.J. Fisher, R.J. Fowler, S.C. Vestal
The architecture of the Eden system
ACM Proc. of the 8th Symposium on Operating System Principles, Pacific Grove, California (Dcember 1981), pp. 148,159

[LEB88] P. Lebee, K.M. Hou, M. Guillemenont, G. Fontenier
*A New Machine Architecture for Distributed Operating Systems*78 Microprocessing and Microprogramming, vol. 22,3 (1988) pp. 187,203

[LEG88] J. Legatheaux, Y. Berbers
La désignation dans les systèmes d'exploitation répartis
Technique et Science Informatique, vol. 7,4 (1988) pp. 359,372

[LEV84] H.M. Levy
Capability-based computer systems
Digital Press (1984)

[LIN76] T.A. Linden
Operating System Structures to Support Security and Reliable Software Comm. ACM vol. 8,4 (1976), pp. 409,445

[LIS79] B.H. Liskov
Primitives for Distributing Computing
7th ACM Symposium on Operating System Principles, Pacific Grove, California, (December 1979), pp. 33,42

[LIS87] B.H. Liskov, D.Curtis, P.Johnson, R.Scheifler
Implementation of Argus
11th ACM Symposium on Operating System Principles, (1987) pp. 111,122

[LIS88] B.H. Liskov *Distributed Programming in Argus*
Communication of the ACM vol.31,3 (March 1988) pp.

300,312

[MET76] R.M. Metcalfe, D.R. Boggs
Ethernet: Distributed Packet Switching for Local Computer Networks
Comm. ACM, vol. 19,7 (July 1976), pp. 395,404

[MEY88] B. Meyer
Object Oriented Software Construction
Prentice Hall International (1988)

[MIL87] B.P. Miller, D.L. Presotto, M.L. Powell
DEMOS/MP: The Development of a Distributed Operating System
Sotf. Practice and experience, vol. 17,4 (april 1987), pp. 277,290

[MUL86] S.J. Mullender, A.S. Tanenbaum
The Design of a Capability-Based Distributed Operating System
the Computer Journal, vol. 29,4 (1986) pp. 289,299

[ORG72] E.I. Organick
The Multics System: An Examination of its Structure
MIT Press (1972)

[PLA88] J.M. Place
Un pré-processeur Modula-2 pour OMPHALE
à paraître

[POW83] M.L. Powell, B.P. Miller
Process migration in DEMOS/MP
Operating Systems Review, vol. 17,5, pp. 110,119

[RAS85] R. Rashid, G. Robertson
Accent: a communication oriented network operating system kernel
Proc. Eighth Symp. on Operating Systems Principles, ACM Operating Systems Review, vol. 15, 5 (December 1981), pp. 64,75

[REN88] R. van Reness, H. van Staveren, A.S. Tanenbaum
Performance of the World's Fastest Distributed Operating System
ACM Operating System Review, vol 22,4 (October 1988) pp. 25,34

[RIT74] D.M. Ritchie, K. Thompson
The UNIX time sharing system
Comm. ACM vol. 19,5 (July 1974) pp. 365,375

[ROZ88] M. Rozier and all
CHORUS Distributed Operating Systems Draft (Septembre 1988)

[SAN86] R. Sandberg
The Sun Network File System: design, implementation and experience
European Unix Systems Users Group (EUUG) Spring Conf. Proc., Florence (april 1986)

[SHA87] M. Shapiro, V. Abrossimov, P. Gautron, S. Habert, M. Mounchilimakpangoumi
SOS: un système d'exploitation réparti fondé sur les objets
Technique et Science informatique, vol. 6, 2 (1987) pp. 166,169

[SCE82] Rapport Sceptre
Rapport BNI no 26/2 (septembre 1982)

[SIL77] A. Silberschatz, R.B. Kieburtz, A. Berstein
Extending Concurrent Pascal to allow dynamic resource management
IEEE Trans Software Eng., vol. SE–3 (May 1977), pp. 210,217

[SOL78] M.H. Solomon, R.A. Finkel
Roscoe: a multi– microcomputer operating system
2nd Rocky Mountain Symposium on Microcomputers, Pingree Park, Colorado, (August 1979), pp. 291,310

[SOL79] M.H. Solomon, R.A. Finkel
The Roscoe distributed operating system
7th ACM Symposium on Operating System Principles, Pacific Grove, California, (December 1979), pp. 108,114

[STE86] M. Stefik, D.G. Bobrow
Object Oriented Programming: Themes and Variations
A.I. Magazine, vol. 6,4 (1986)

[STR81] B. Stroustrup *Long Return: A Technique for Improving the Efficiency of Inter–module Communication*
Software–Practice and Experience, vol. 11 (1981), pp.

131,143

[SUN86a] Sun Microsystems, Inc. *Networking guide: XDR Protocol Specification*
800–1324–03 (February 1986)

[SUN86b] Sun Microsystems, Inc. *Networking guide: IPC primer*
800–1324–03 (February 1986)

[TAN81a] A.S. Tanenbaum, S.J. Mullender
An overview of the AMOEBA distributed operating system
Operating Systems Review, vol 15, 3, (July 1981), pp. 51,64

[TAN81b] A.S. Tanenbaum
Computer Networks
Prentice–Hall International (1981)

[TAN85] A.S. Tanenbaum, R. Van Renesse
Distributed Operating System
ACM Computing Surveys, vol. 17,4 (decembre 1985)

[TAN86] A.S. Tanenbaum, S.J. Mullender, R. Van Renesse
Using sparse capabilities in a distributed operating system
Proc. of the 6th International Conference on Distributed Computer Systems, IEEE, New–York, 1986, pp. 558,563

[TAN87] A.S. Tanenbaum
Operating Systems: design and implementation
Prentice–Hall International (1987)

[WIR83] N. Wirth
Programming in Modula–2
Springer–Verlag, Berlin (1982)

[YON87] A. Yonezawa, M. Tokoro
Object Oriented Concurrent Programming
The MIT Press (1897)

[ZIM81] H. Zimmerman, J.S. Banino, A. Caristan, M. Guillemont, G. Morrisset *Basic concepts for the support of distributed systems: the CHORUS approach*
2nd International Conference on Distributed Computing Systems, Versailles, France (April 1981),

pp 60,66

[ZWA84] W. Zwanenepoel, K. Lantz
PERSEUS: Retrospective on a Portable Operating System
Software–Practice and Experience, vol. 14,31 (1984) pp. 31,48