Numéro d'ordre: 324

ANNEE: 1989

USTL FLANDRES ARTOIS — IFL U.A. 369 C.N.R.S. — CNRS

LABORATOIRE D'IN RMATIQUE FONI MENTALE DE LILLE

50376
1989
41

THÈSE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES
DE LILLE FLANDRES ARTOIS

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

Une implantation de l'architecture de système réparti OMPHALE.

par

Jean-François Méhaut

Thèse soutenue le 10 Février 1989, devant la commission d'examen

Membres du Jury:

Président	:	M. Dauchet,	Professeur USTLFA
Directeur	:	E. Delattre,	Maître de Conférences USTLFA
Rapporteurs	:	C. Carrez,	Professeur CNAM Paris
	:	B. Toursel,	Professeur USTLFA
Examinateur	:	S. Tison,	Maître de Conférences USTLFA

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

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

Remerciements

Je remercie les membres du jury:

- Max Dauchet, Professeur à l'U.S.T.L.F.A., directeur du L.I.F.L., pour avoir accepté la présidence de ce jury. Je le remercie aussi, pour d'abord m'avoir incité à entreprendre la rédaction de cette thèse, puis ensuite pour ses nombreux encouragements.
- Christian Carrez, Professeur au C.N.A.M. de Paris, pour m'avoir dans un premier temps accordé sa confiance en dirigeant cette thèse, puis après son départ au C.N.A.M., pour avoir accepté de rapporter cette thèse. Je le remercie aussi pour ses remarques sur le document.
- Bernard Toursel, Professeur à l'E.U.D.I.L, pour avoir rapporté cette thèse.
- Eric Delattre, Maître de conférences à l'U.S.T.L.F.A., pour avoir dirigé cette thèse. Ses conseils, critiques constructives m'ont permis d'achever ce travail. Ses nombreuses remarques m'ont particulièrement aidé dans la rédaction de cette thèse.
- Sophie Tison, Maître de conférences à l'U.S.T.L.F.A., pour avoir examiné cette thèse avec le plus grand soin.

Je remercie également:

- Jean-Marc Geib et Jean-Marie Place, membres de l'équipe OMPHALE, pour les longues et fructueuses discussions que nous avons eues.
- L'office des pupilles de la nation et plus particulièrement Monsieur Coudert qui m'ont toujours apporté une aide matérielle et morale. Cette thèse est aussi le résultat de leur travail.
- Nathalie Komorowski, pour avoir eu la gentillesse et la patience de relire ce document.
- Lauren Arango, Professeur à l'Université d'OHIO, pour avoir corrigé le résumé en Anglais.
- Henri Glanc, pour avoir assuré la reproduction de cette thèse.

Table des Matières

Avant-propos

OMPHALE est un projet 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. OMPHALE est un système d'exploitation réparti. L'équipe OMPHALE a été dirigée par Christian Carrez (Professeur au C.N.A.M de Paris) puis par Eric Delattre (Maître de conférences à l'U.S.T.L.F.A).

L'équipe OMPHALE comprend également:

- Jean-Marc Geib, qui a établi les spécifications de l'architecture OMPHALE. Le lecteur consultera sa thèse [GEI 89] où sont largement détaillés et justifiés les choix de base de l'architecture OMPHALE. Jean-Marc Geib compare les choix OMPHALE avec ceux effectués sur les autres systèmes répartis.
- Jean-Marie Place qui, avec Eric Delattre, a conçu une nouvelle architecture dédiée à OMPHALE [DEL85b] [DEL 86] [DEL88] . Cette architecture comprend deux processeurs spécialisés. En outre, elle possède un changement de contexte rapide et une circuiterie de partage mémoire entre les deux processeurs.
  Jean-Marie Place a également défini un langage de programmation adapté à l'architecture OMPHALE.

OMPHALE est une architecture de système réparti permettant la communication asynchrone par messages entre les processus (**modules**). Le module OMPHALE fait la synthèse de trois approches qui font qu'un module est à la fois un processus, un serveur et un objet. Le nommage et la protection sont assurés par un mécanisme de capacités (**liens**). L'architecture OMPHALE est structurée en quatre couches:

- **Matériel**
  Le niveau Matériel de l'architecture OMPHALE se compose d'un réseau local auquel sont connectés un ensemble de sites. Un site est équipé d'un voire plusieurs processeurs, de la mémoire et des périphériques. La mémoire et les périphériques ne sont accessibles que par le (ou les) processeur du site. Au niveau de l'architecture OMPHALE et des systèmes répartis, nous pouvons remarquer l'absence de mémoire commune à l'ensemble des sites.
- **NERF**
  Un noyau NERF s'exécute sur chaque site de l'architecture. Il assure l'enchaînement des exécutions des modules du site. Avec l'aide de la station de transport, il réalise les transferts de messages entre les modules.
- **CORTEX**
  Les services systèmes de haut-niveau (gestion mémoire, création/destruction de modules, gestion fichiers) sont rendus par des modules de la couche CORTEX. L'avantage est alors de ne laisser au niveau du noyau NERF qu'un minimum de fonctionnalités. Les modules de la couche CORTEX sont répartis sur les différents sites du réseau.

- **Application**
  Les processus de l'application composent la couche application. Cette couche est répartie sur l'ensemble des sites de l'architecture.

Le **chapitre 1** présente les concepts de base de l'architecture OMPHALE tels que le module, le service et les liens.

Notre travail consistait à évaluer les problèmes de l'implantation de l'architecture OMPHALE. Dans un premier temps, nous avons évalué l'architecture matérielle du SPS7 de la compagnie BULL [BUL 86a]. La SPS7 résulte de la commercialisation par la compagnie BULL de la SM90 développée par U. Finger [FIN 81]. Cette architecture matérielle est particulièrement modulaire et souple. Il est en effet très facile d'ajouter ou supprimer des modules (processeurs). Le **chapitre 2** présente les principales caractéristiques de la SPS7.

Dans l'étape suivante, nous avons réfléchi sur l'implantation du noyau NERF. Plusieurs solutions ont été envisagées:

- **Développement sur machine nue**
  Il s'agit de reconstruire un système complet en partant du matériel. L'effort de développement est important et nos moyens en main d'oeuvre sont limités. L'équipe OMPHALE n'est en effet composée que de 5 enseignants-chercheurs.
  D'autre part, la documentation du matériel fournie par le constructeur s'est révélée insuffisante.
  L'avantage de développer sur machine nue est de pouvoir maitriser complétement les mécanismes mis en oeuvre. L'évaluation des performances du système est facilitée.
- **Développement sur un système existant**
  Il s'agit en fait d'utiliser les fonctionnalités d'un système existant et ce pour alléger le volume de programmation. La BULL SPS7 propose deux systèmes:
  - **UNIX**
    Le système UNIX [RIT 74] de la SPS7 s'appelle SPIX [BUL 86c]. C'est un dérivé d'UNIX système V. En utilisant les processus UNIX et les extensions du système V (les IPC inter-process communication), il est possible d'implanter l'architecture OMPHALE. Cette solution présente l'avantage de la portabilité de l'architecture. Cependant, on peut douter de l'éfficacité du système. En effet, la programmation OMPHALE requiert un grand nombre d'objets donc de processus. SPIX montre des signes de faiblesse quand le nombre de processus devient grand (à partir de 30 processus). Ces problèmes sont en partie dus à l'activité de va-et-vient (swapping) et aux faibles performances des disques.
  - **SPART**
    SPART [ALB 85] [BUL 86b] est un système temps-réel. Ses principes sont largement inspirés du rapport SCEPTRE [SCE 82] [VOJ 84] qui proposait une normalisation des exécutifs temps-réels. Le noyau SPART définit un ensemble d'objets. Chaque type d'objet est déclaré dans une agence. Les primitives d'utilisation de l'objet sont décrites dans les agences. Pour accroitre la protection, les primitives sont exécutées en mode système.

Le noyau est **extensible** par adjonction de nouvelles agences. Cet aspect d'extensibilité est important car on peut intégrer à ce noyau les fonctionnalités de notre noyau NERF. Nous avons donc choisi d'implanter le noyau NERF à partir du noyau SPART. Le **chapitre 2** décrit les éléments importants du noyau SPART.

Le **chapitre 3** décrit l'intégration des fonctionnalités OMPHALE au noyau SPART. Les structures de données du noyau NERF sont d'abord décrites. Les primitives du noyau NERF y sont ensuite détaillées. Chaque primitive est présentée de la manière suivante:

- Une spécification de la primitive,
- Son implantation c'est à dire le texte source en langage C [KER 78] de la primitive,
- Des commentaires aidant à la compréhension du texte source.

Le **chapitre 4** reprend des applications développées sur l'architecture OMPHALE. Ces programmes ont permis de contrôler la validité du noyau NERF mais également de définir une méthodologie de programmation. Le **Chapitre 4** décrit les exemples suivants:

- La structure de donnée Pile est implantée à partir d'un module OMPHALE.
- Des modules de la couche système CORTEX tels que le gestionnaire de noms symboliques ou le créateur de modules sont décrits.
- Une solution au problème "des philosophes et des spaghettis" est également décrite. Elle permet de mettre en évidence que l'envoi de message est également un outil de synchronisation.

Cette étude se termine par une évaluation de l'implantation de l'architecture OMPHALE. Les choix d'implantations sont analysés et comparés avec ceux effectués sur d'autres implantations de systèmes.

Chapitre 1
L'architecture OMPHALE

1.1 Les concepts de base du système OMPHALE

OMPHALE [DEL85a] [GEI 89] est une architecture de système réparti. Le schéma ci-dessous décrit la configuration matérielle de ce système réparti.

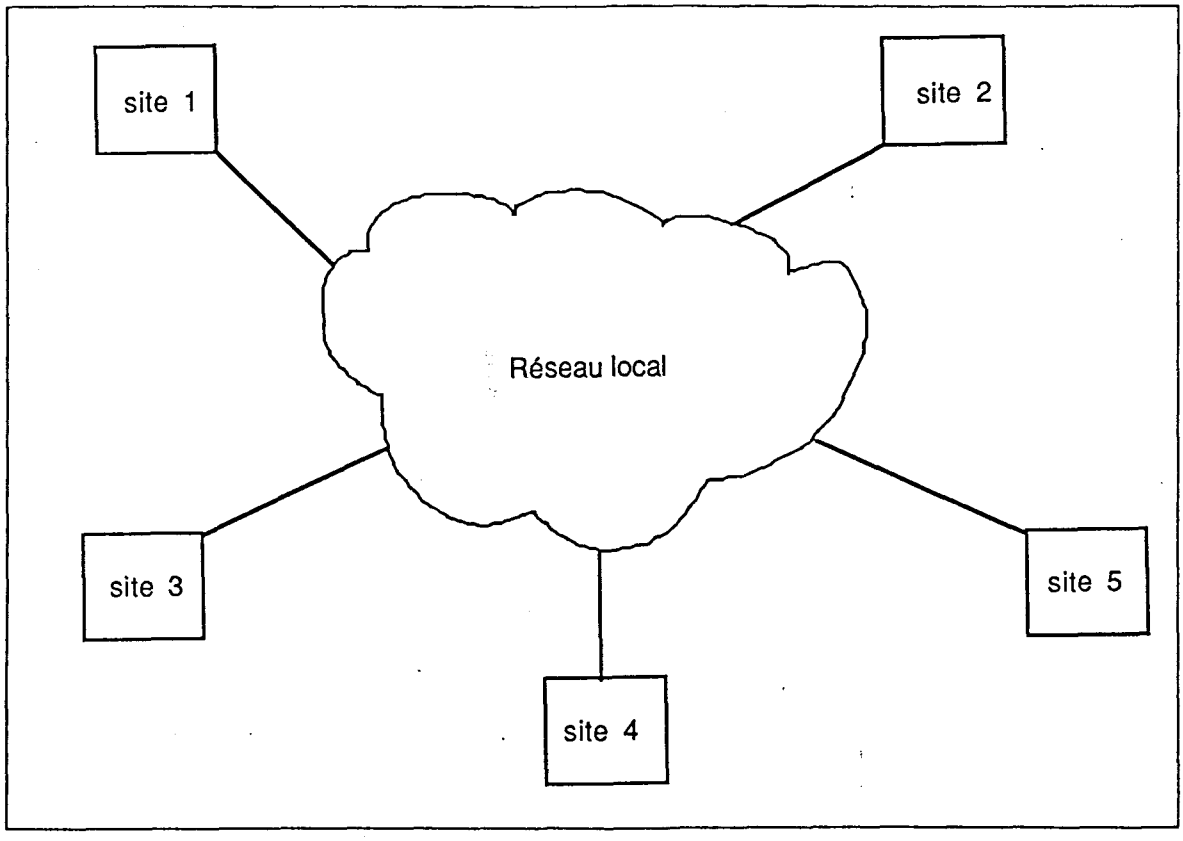


Figure 1. L'architecture des systèmes répartis

Cette architecture se compose des éléments suivants:

- **Un réseau local**
  Le réseau local posséde les propriétés suivantes:
  - **Haut-débit**
    Les caractéristiques physiques du réseau doivent fournir de grandes vitesses de transfert de données.

– **Fiabilité**
Ce réseau doit offrir une grande fiabilité pour les transferts.

La topologie du réseau n'a que peu d'importance dans nos travaux. Nous supposons seulement que le maillage du réseau est complet c'est à dire que tout ordinateur connecté au réseau peut communiquer avec n'importe quel autre ordinateur du réseau.

- **Les sites**
Pour désigner un ordinateur du réseau, la terminologie des systèmes répartis [COR 81] propose le terme **site**. OMPHALE reprend donc ce terme. Un site est un ordinateur complet comprenant:

  – son processeur (Certains sites pourront être équipés de plusieurs processeurs),

  – sa mémoire,

  – ses périphériques.

Les architectures distribuées ont de nombreux avantages [KRA 87] [GEI 89]:

- **Partage de ressources**
La fonction principale d'un système est de partager les ressources (matérielles ou logicielles) entre les différents utilisateurs. Sur une architecture répartie, les ressources et les utilisateurs sont également répartis sur le réseau. Pour utiliser une ressource, il n'est pas nécessaire que l'utilisateur de la ressource se trouve sur le même site que celle-ci. L'utilisateur n'a pas à se préoccuper de la localisation des ressources. Le réseau est **transparent** c'est à dire que l'utilisation d'une ressource locale ou distante se fait de la même manière.

- **Parallélisme**
Les processus résidant sur des sites différents s'exécuteront en parallèle. Il est nécessaire que le noyau du système et la programmation des applications prennent en compte les possibilités de parallélisme de l'architecture.

- **Disponibilité**
La panne d'un site n'entraine pas la paralysie du système. Les possibilités de fonctionnement en mode **dégradé** sont plus importantes. En particulier, la répartition des ressources sur l'ensemble des sites augmente les taux de disponibilité. Il est donc judicieux de répartir équitablement les ressources sur les différents sites de la configuration.

- **Extensibilité**
Elle consiste à ajouter de nouveaux éléments. L'ajout d'un site consiste d'abord à le connecter au réseau et ensuite à mettre à jour des informations sur les autres sites.

L'absence de mémoire commune nécessite de nouvelles solutions aux problèmes classiques posés pour la conception de systèmes. En particulier, les solutions aux problèmes de synchronisation sont fondamentalement différentes en contexte centralisé et distribué [AND 83] [RAY 84]. Sur les architectures réparties, elles reposent sur les duplications d'information. Se posent alors des problèmes de mise à jour de ces informations. Le paramètre temps et sa prise en compte conduit, par exemple à des solutions avec estampillage [LAM 78].

1.1.1 Programmation parallèle et communication

Un programme parallèle se compose de processus communiquant et se synchronisant. Deux points sont à définir:

- La structuration du programme en terme de processus,
- Les points de synchronisation et de communication entre les processus de l'application.

1.1.1.1 Structure multi-processus

Le concept de programmation structurée est apparu dans les années 1960. Des langages traditionnels tels que ALGOL 60 ou même PASCAL [WIR 71] permettent l'écriture de programmes structurés purement séquentiels. Très vite apparut la nécessité d'écrire des programmes parallèles. En effet, il semblait naturel de découper le programme en activités indépendantes ou quasi-indépendantes. Cette indépendance va de pair avec la notion de parallèlisme d'exécution des activités. On appelera ces activités des processus ou des tâches. Un programme parallèle se compose d'un ensemble de processus s'exécutant en parallèle (ou en pseudo-parallélisme).
Un processus comprend des données locales qu'il est le seul à pouvoir manipuler. Un processus ne peut accéder directement aux données locales d'un autre processus. La communication d'informations entre les processus de l'application est indispensable. Au niveau d'une architecture à mémoire commune, une solution est d'implanter des données globales et partagées. Les outils de synchronisation permettent de contrôler l'intégrité de ces données. Dans un contexte réparti, l'absence de mémoire commune rend impossible le stockage de données globales. Les primitives de communication permettent alors le partage d'information entre les processus.
Le paragraphe **Communication** décrira les principales techniques de communication inter-processus.
Pourquoi cette nécessité de structuration d'un programme en terme de processus [CHE 82]?

- **Temps-réel**
  C'est initialement dans l'activité temps-réel que l'on s'intéressa au découpage de programmes en termes de processus. En effet, le développement d'applications temps-réel posait quelques problèmes.
  - Certaines séquences d'instructions devaient être exécutées avec une priorité plus importante. Il était difficile de faire apparaître cette contrainte au milieu d'un programme.
  - Certaines instructions nécessitaient une exécution en mode système.
  - Certaines instructions devaient être exécutées périodiquement.

  La solution retenue fut de regrouper dans un même processus les instructions devant s'exécuter sous les mêmes contraintes (priorité, mode). Des caractéristiques d'exécution sont alors affectées globalement au niveau de chaque processus.
- **Outil de structuration**
  A l'instar des procédures ou fonctions, on s'aperçut très vite que la notion de processus était un outil qui aidait à la structuration des programmes. Le découpage du programme en pro-

cessus augmente sa lisibilité et sa compréhension. Les procédures de maintenance sont facilitées.

- **Parallélisme**
  La programmation séquentielle ne répondait plus aux besoins exprimés avec les nouvelles architectures. Une des principales caractéristiques de ces nouvelles architectures est le parallélisme. Beaucoup d'architectures sont maitenant multi-processeurs ou même multi-ordinateurs (architectures réparties). Il est nécessaire que la programmation tienne compte efficacement de telles architectures. La solution est alors de découper le programme en processus. Les processus du programme s'exécuteront sur les différents processeurs de l'architecture.

- **Coûts de développement**
  Le coût du matériel a tendance à baisser par rapport à celui du logiciel. La structuration en terme de processus diminue le coût de développement. La production du code des processus (compilation, édtion de liens) est réalisée indépendamment pour chaque processus. La modification d'un processus n'entraîne pas la recompilation des autres processus du programme.

Cette structuration en termes de processus a conduit des nouveaux langages (ADA [DoD 83], OCCAM [INM 84], MODULA-2 [WIR 83]) à introduire le concept de tâche (processus).

1.1.1.2 Communication

La programmation de programmes parallèles montre que les processus ont des inter-actions entre eux. Ces inter-actions se résument ainsi:

- **Synchronisation**
  Le problème de l'accès en exclusion mutuelle illustre ce propos. Un certain nombre de processus ont accès (lecture, écriture) à une donnée partagée. L'intégrité de cette donnée est menacée si ces accès ne sont pas contrôlés. Le noyau du système propose des outils (sémaphores, moniteurs, régions ...) qui permettent la synchronisation de processus.

- **Communication**
  Les processus éprouvent le besoin de communiquer. Un processus P a besoin d'une information détenue par le processus Q. Le processus Q doit **communiquer** cette information au processus P. Les systèmes d'exploitation proposent des outils de communication.

Les outils de synchronisation et communication sont implantés à partir de mécanismes matériels de bas niveau. Ils sont donc fortement dépendants de l'architecture matérielle:

- **Systèmes à mémoire commune**:
  La mémoire commune contient les informations à partager entre les processus. Il s'agit alors de contrôler les accès concurrents aux objets en mémoire commune. Les solutions retenues au niveau matériel sont du type "test and set" ou basées sur l'indivisibilité des opérations de lecture-écriture.

- **Systèmes répartis:**
  L'absence de mémoire commune rend impossible les solutions des systèmes centralisés. L'envoi de messages est l'outil de communication et de synchronisation privilégié des systèmes répartis ([LIS 79], [COR 81], [GUI84a], [RAS 84]...). Si le processus P désire communiquer une donnée D à un processus Q, P doit émettre un message destiné à Q. Le contenu du message sera la donnée D. Il existe deux formes d'envoi de messages:

  - **synchrone** :
    Le processus émetteur est bloqué jusqu'à la consommation du message par le récepteur. La synchronisation par envoi synchrone de messages est fonctionnellement équivalente au mécanisme de rendez-vous tel qu'il est proposé dans ADA [DoD 83].
    OCCAM [INM84a] propose les canaux de communication [HOA 78]. Les canaux de communication sont particulièrement adaptés à l'architecture cible du langage OCCAM: Le transputer d'Inmos [INM84b].
    Les communications synchrones ont l'inconvénient de limiter le parallèlisme. Le processus émetteur est bloqué tant que son message n'est pas consommé. Le nombre de processus prêts diminue de un pendant le blocage de l'émetteur.

  - **asynchrone** :
    Le processus émetteur n'est pas bloqué par une émission de messages. L'activité du processus émetteur reprend dès la prise en charge du message par le système de transport. Les messages émis, et qui n'ont pas encore été traités, sont stockés dans des boîtes aux lettres qu'on appelle aussi des ports de communication ([GUI84a], [RAS 84]). L'envoi asynchrone de message demande donc un espace mémoire supplémentaire pour les messages en attente de traitement.
    L'envoi asynchrone de messages favorise la parallélisme (Le nombre de processus prêts ne diminue pas sur un envoi de messages).
    L'envoi synchrone peut être simulé par envoi asynchrone; C'est à la charge du programmeur de gérer un dialogue Question-Réponse entre l'émetteur et le récepteur.

  L'envoi de messages présente les avantages suivants:

  - L'envoi de messages permet à la fois la communication et la synchronisation de processus.
  - L'envoi de message peut aussi bien être implanté sur des architectures distribuées que centralisées.

1.1.2 Modèle objet

La méthodologie de programmation orientée-objet connaît un grand succès actuellement. Cette méthodologie s'applique parfaitement à la conception des systèmes d'exploitation [JON 79]. En effet, un système informatique peut être décrit comme un ensemble de ressources. Un objet est alors associé à chaque ressource. Le noyau de tels systèmes d'exploitation doit donc gèrer un ensemble d'objets.

Chaque objet (ressource) se caractérise par:

- Une valeur c'est à dire l'état de l'objet. Cette valeur est encapsulée dans l'objet. Sa modification n'est possible que par les opérations définies sur l'objet.
- Pour utiliser un objet, il est nécessaire d'utiliser les primitives (méthodes) définies sur l'objet. Ces primitives ont pour conséquence de modifier l'état de cet objet.

On peut citer en exemple le fichier. Avec ce modèle, le fichier sera modélisé par un objet. L'objet fichier se caractérise par:

- L'état du fichier (Ouvert, Ferme, position de la fenêtre de lecture/écriture, état du tampon).
- Les méthodes de l'objet fichier seraient:
  - **Open**
    Opération d'ouverture du fichier.
  - **Read/Write**
    Opérations de lecture écriture.
  - **Close**
    Opération de fermeture du fichier.

On peut faire quelques remarques sur ce modèle:

- **Structuration en couches**
  Il est maintenant couramment admis qu'un système d'exploitation peut être structuré en couches (THE [DIJ68a], XINU [COM 83], MINIX[TAN 87]) Cette méthodologie n'est pas incompatible avec le modèle objet. Chaque couche définira des objets qui utiliseront les objets des couches inférieures. Avec ce modèle, au fur et à mesure que l'on remonte dans la hiérarchie, les couches proposent des objets plus complexes.
- **Richesse**
  La richesse d'un système peut se mesurer avec la variété des objets qu'il propose. Cette richesse est également fonction des méthodes proposées par chaque objet.
- **Capacité**
  Les caractéristiques matérielles de la machine sont limitées. La capacité d'un système peut donc se mesurer avec le nombre d'objets que l'architecture est capable de supporter.

1.1.2.1 Messages

Pour exécuter une primitive sur un objet, il est nécessaire d'envoyer un message à cet objet. Ce message se compose de:

- **Sélecteur**
  L'envoi de message se fait en désignant un objet. Le sélecteur permet d'indiquer la méthode à exécuter sur cet objet. Pour l'exemple fichier, le selecteur indiquera l'une des méthodes (Open, Read, Write ou Close) sur le fichier.

- **Paramètres**
  Chaque méthode a des paramètres. Les paramètres de la méthode sont placés dans le contenu du message. Dans l'exemple fichier, le message contiendra la donnée à écrire sur le fichier.

L'abstraction est un concept important de la programmation [BIS 86]. L'envoi de message s'intègre parfaitement à ce concept. Un objet se compose de deux parties:

- **Spécification**
  Une spécification d'objet doit fournir les informations concernant les méthodes (opérations) proposées sur l'objet. La spécification répond à la question suivante: "Que peut-on faire avec cet objet?"
- **Implantation**
  L'implantation d'un objet consiste à décrire les structures de données utilisées par l'objet. Le corps des méthodes de l'objet (algorithme) se trouve dans l'implantation. L'implantation répond à la question suivante:" Qu'utilise-t'on comme structure interne et comment le fait-on?". Toute entité déclarée dans une implantation est locale à cette implantation.

Pour envoyer un message, il suffit de connaître la spécification de l'objet. (liste des méthodes définies par l'objet). La modification d'une implantation n'entraine aucune modification des utilisations de l'objet.

ADA [Dod 83], MODULA-2 [WIR 83] proposent un mécanisme de spécification et d'implémentation sur les paquetages (modules). Ce mécanisme permet l'implantation de types de données abstraits [BIS 86].

1.1.3 Modèle Client-Serveur

Un système informatique peut être vu sous la forme d'un ensemble de ressources. Le système d'exploitation a alors pour fonction d'allouer les ressources aux usagers [KRA 85]. La structuration du système peut alors être bâtie autour du modèle Client-Serveur. Ce modèle fait apparaître deux entités que nous présentons:

- **Serveur**
  Un processus **Serveur** est chargé de fournir des **services** pour le compte de processus **Client**. Un processus **Serveur** est généralement associé à chaque ressource. Le processus **Serveur** posséde deux états:
  - **Actif**
    Le serveur a reçu une requête de service. L'appel d'une requête peut être implanté par un envoi de messages. Le message contient alors les paramètres du service ainsi que le nom de la requête. Le **serveur** exécute le service. Ce service peut nécessiter le retour d'une réponse au processus client (envoi de message). Dans ce cas, l'identité du processus **Client** doit figurer dans la requête. Sitôt la requête exécutée, le **Serveur** se mettra en attente d'une nouvelle requête.

- **Attente**
  Le processus **Serveur** est dans l'attente d'une requête provenant des processus **Client**. Le processus **serveur** reste en **Attente** tant qu'aucune demande ne provient des processus **Client**. Au niveau du système, le processus **serveur** est dans l'état bloqué. Ce qui n'apporte aucune charge supplémentaire au processeur.

Pour satisfaire une requête, un processus **serveur** pourra faire si nécessaire appel à d'autres processus **Serveur**. Il se comportera comme client vis-à-vis de ces serveurs. La réponse faite au client, pourra être émise par l'un des serveurs qui sous-traite le service: C'est le long-retour [STR 81].

- **Client**
  Pour utiliser une ressource, le processus **Client** doit impérativement adresser une requête à un processus **Serveur**. C'est le serveur qui exécutera la requête. Si la requête nécessite une réponse, le client pourra placer son nom dans le message de requête.

Ce modèle posséde les qualités suivantes:

- **Partage de ressources**
  Ce modèle règle simplement le partage de ressources entre plusieurs processus. Le **serveur** gère sa politique d'utilisation de la ressource en validant ou en invalidant ses services.
- **Abstraction**
  La ressource n'est visible que du processus **Serveur**. Les détails de son utilisation ne sont visibles que du serveur. Une modification du fonctionnement interne de la ressource ne modifie en rien les processus **Clients**.
- **Noyau minimal**
  Le modèle Client-Serveur est utilisé pour la construction de systèmes d'exploitation. En exemple, on peut citer MINIX [TAN 87] qui a défini un serveur de fichiers et un serveur mémoire. Ce modèle permet de sortir du noyau des fonctionnalités et de les affecter à ces processus serveurs. La taille du noyau est ainsi considérablement réduite.
- **Souplesse**
  L'ajout d'une nouvelle ressource consiste à créer un nouveau processus **Serveur**. Dans le cas de ressources banalisées, le processus **Client** s'adresse à un pool de processus **Serveur**. La requête est satisfaite par l'un des processus **Serveur**.
- **Intégration aux architectures réparties**
  Ce modèle s'adapte parfaitement aux systèmes répartis. Le noyau des systèmes répartis assure la transparence du réseau. Ce qui a pour conséquence dans le modèle **Client-Serveur** de banaliser la localisation des **Serveurs**. Les **Clients** n'ont ainsi donc pas à se préoccuper de la localisation des **Serveurs** sur le réseau.

1.1.4 Le module OMPHALE

OMPHALE fait une synthèse des trois approches précédentes. OMPHALE intègre les notions de *processus*, *serveur* et *objet* dans une entité unique appelée **module**. Le **module** est à la fois:

- un *processus* séquentiel qui est un outil de décomposition du programme. La synchronisation et la communication avec les autres modules du programme se fait par envoi asynchrone de messages. Le module réside complétement sur son site de création jusqu'à sa destruction.
- un *Serveur*, en ce sens où le module peut être vu comme un processus *Serveur*. Il propose un ensemble de services à ceux qui le connaissent (*Clients*). Pour exécuter un service, le module *Client* doit envoyer un message (requête) au module *Serveur*.
- *Objet* au sens de l'approche orientée-objet. Un module posséde un état et des services (méthodes, procédures). Pour exécuter une procédure par un module, un message doit être envoyé à ce module.

1.1.5 La structuration du système

La structuration des systèmes d'exploitation en couches est maintenant passée dans les mœurs des concepteurs. De ce fait, OMPHALE est structuré en quatre couches:

- **Matériel**
  La couche matérielle se compose d'un réseau local et d'un ensemble de sites connectés à ce réseau.
- **Noyau NERF**
  Le noyau NERF s'exécute sur chacun des sites de la configuration. Il assure la gestion des modules du site. Le noyau NERF gére les communications locales de messages. Les communications distantes (vers d'autres sites) se font en collaboration avec la station de transport.
  Le noyau définit un mécanisme de nommage et de protection des modules autour de la notion de **lien** ([FAB 74], [BAS 77], [DEL 79], [GEI 89]). (**capacités** ou **liens**).
- **CORTEX**
  Collection d'outils systèmes répartis.
  Cette couche se compose d'un ensemble de modules systèmes. Elle offre des services systèmes de haut-niveau tels que la gestion mémoire, le système de fichiers, le nommage symbolique, le création dynamique de modules etc... Ces modules se comportent comme des modules serveurs. Cette couche a l'avantage de sortir du noyau certaines fonctionnalités et de proposer un noyau **minimal**. Cette couche est répartie sur les différents sites.
- **Application**
  Elle se compose d'un ensemble de modules répartis sur le réseau. L'exécution des différents modules, la communication entre ces modules contribuent à l'exécution de l'application.

1.2 Les messages OMPHALE

Les modules OMPHALE communiquent par envoi de messages. L'envoi de messages est **asynchrone** c'est à dire que l'émetteur n'est pas bloqué par l'émission. L'envoi de messages est **anonyme**. L'identité du module émetteur ne figure pas dans le message. Le message OMPHALE se compose:

- **nom de module:**
  Il permet d'identifier le module à qui on envoie le message.
- **nom de service** (sélecteur):
  Il permet d'identifier le service (la méthode) dont on demande l'exécution.
- **contenu du message:**
  Le message OMPHALE comprend:
  - Une suite d'octets que le noyau NERF n'interpréte pas. La longueur de cette suite (du message) est variable et fonction de l'application. Le passage d'adresses (pointeurs) dans les messages n'a aucun sens sur une architecture sans mémoire commune. Le contenu d'un message OMPHALE ne comprend que des valeurs.
  - Des noms de modules (**liens**). Le transfert de liens dans les messages réalise une édition dynamique de liens [COR 65] entre les modules OMPHALE. Le lien est une structure de donnée protégée du noyau. Le noyau définit donc des primitives de manipulations de liens dans les messages. Le module peut ainsi placer des liens dans ses messages sans accéder à la structure Lien.
    L'envoi de messages OMPHALE est **anonyme**. L'identité du module émetteur ne figure pas automatiquement dans le message. C'est au module à placer son identité (*Myself*) dans le message si cela est nécessaire.

On peut remarquer que le message OMPHALE présente des similitudes avec le message de l'approche objet.

1.3 Le transfert de messages

Cette section décrit les principales étapes du transfert de message de l'émetteur à son récepteur.

1. **Préparation des messages**
   Chaque module OMPHALE posséde un tampon de messages qu'on appelle la zone d'émission (ZE). Cette zone contient les messages que le module désire envoyer. Cette zone est gérée par le noyau NERF. Le noyau NERF fournit des primitives permettant d'ajouter ou de retirer des messages de cette zone. La zone d'émission contient à la fois des messages vers les modules voisins (sur le même site) ou vers des modules distants (sites différents). Il n'y aucune différence entre la préparation d'un message vers un module voisin ou vers un module distant.
   Pour illustrer cette phase, nous prenons en exemple le système postal. Chloé organise une fête pour son anniversaire. Elle utilise le système postal pour ses invitations. Pour chaque invité, elle envoie une lettre d'invitation. Elle place les lettres dans un sac en vue de le porter à la poste. Pour Chloé, envoyer une lettre à une amie habitant la même ville qu'elle ou une amie habitant dans une ville voisine se fait de la même façon. Tant que le sac se trouve chez elle, elle a le loisir d'ajouter de nouvelles invitations ou d'en retirer.
2. **Emission des messages**
   A son initiative, le module prévient le noyau que les messages de la zone d'émission peuvent être émis. Le noyau prend alors en charge les messages de la ZE. Pour en revenir aux lettres

de Chloé, cette phase correspond au dépôt du sac à la poste. A partir de ce moment, Chloé n'a plus aucun moyen d'accéder aux lettres du sac.

3. **Acheminement des messages**
   L'acheminement des messages se fait par le système de transport. Deux cas sont à étudier suivant la localisation des modules émetteurs et récepteurs:

   - **Voisins**
     L'émetteur et le récepteur du message résident sur le même site. Le message est directement envoyé au module destinataire.

   - **Distants**
     L'émetteur et le récepteur du message ne sont pas sur le même site. Le message est passé à un module du CORTEX appelé **station de Transport.** Ce module va assurer le transfert du message vers le site du module destinataire. Le transport des messages distants est assuré à l'extérieur du noyau.

   Dans notre exemple, les lettres de Chloé sont arrivées au centre de tri postal. Les lettres sont triées. Les invitations à destination des amies habitant des villes voisines sont transmises au service ferroviaire. Les invitations pour les amies habitant la même ville sont transmises directement.

4. **Dépôt des messages**
   Chaque module OMPHALE posséde une zone de stockage des messages reçus. C'est la zone de réception (ZR). Le système de transport y dépose les messages que le module a reçu. On appelera dépôt d'un message le moment où le message est déposé dans la zone de réception du module. Cette zone contient donc les messages que le module n'a pas encore traités.
   Sur l'exemple, c'est le moment où la lettre de Chloé arrive dans la boîte aux lettres d'une de ses amies.

5. **Réception du message**
   La réception n'est effective que lorsque le noyau NERF extrait de la zone de réception un message. C'est la réception du message. Le retrait d'un message est toujours postérieur à sont dépôt. Pour en revenir à l'exemple de Chloé, c'est le moment où une amie de Chloé ouvre la lettre et la lit.

1.4 La notion de service

Le module OMPHALE peut être perçu comme un processus **serveur.** Un processus *Serveur* définit un certain nombre de *services* qu'il propose aux modules *Clients*. Pour utiliser un service, le *Client* va adresser une requête de service au *Serveur*. Cette requête se compose de:

- le sélecteur de service. Il permet d'indiquer le service qui doit être activé.
- le contenu du message comprenant les paramètres du service.

Une requête adressée à un module **Serveur** se concrétise par un message.

Chaque module OMPHALE posséde une zone de réception (ZR). La ZR est locale au noyau NERF. A chaque service du module, on associe une file d'attente (qu'on appéllera encore **quai**) des messages arrivés sur ce service. Cette file (quai) est gérée en FIFO.
Pour chaque file, un indicateur précise son status:

- **Ouverte**
  Le module OMPHALE accepte de traiter un message de cette file.
- **Fermée**
  Le module OMPHALE refuse de traiter les messages pouvant se trouver sur cette file.

Pour chaque module on gère un ensemble de files d'attente de messages. Cet ensemble constitue la zone de réception du module.

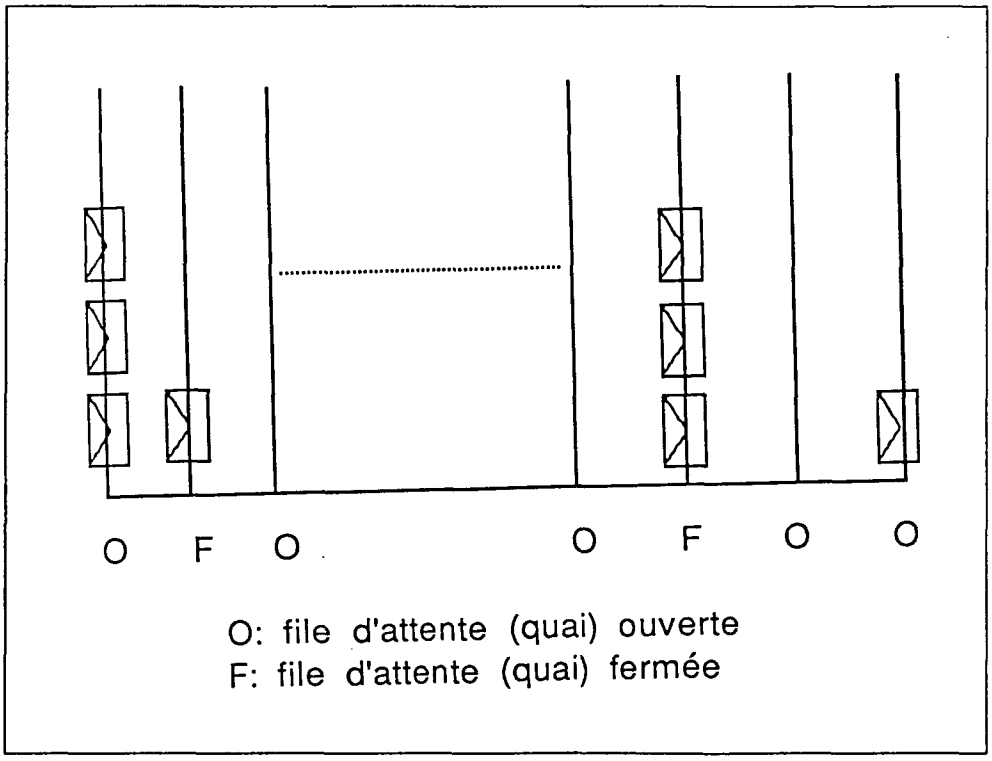


Figure 2. La zone de réception

Lorsqu'un message arrive au module, le noyau NERF ajoute ce message sur la file d'attente associée au nom de service du message. C'est le **dépôt** du message. Un message est extrait de cette structure (d'une des files ouvertes) pendant la phase d'attente du module: C'est la **réception** du message.

1.5 Schéma d'exécution

Le module OMPHALE posséde deux états:

- **Actif**
  Le module se trouve dans une phase d'activité. La réception d'un message a provoqué le réveil du module (passage de l'état attente à l'état actif).
- **Attente**
  Le module est dans une phase d'attente. Cette phase d'attente dure tant qu'il n'y a aucune réception de messages.

La vie du module (depuis sa création jusqu'à sa destruction) se compose d'une suite de phases (Activité, Attente).

1.5.1 Phase d'activité

Une phase d'activité consiste à:

- Récupérer les paramètres qui sont dans le message. En fonction de ces paramètres, le module effectue certains calculs.
- Emettre des messages vers d'autres modules. Pour exécuter le service, le module a besoin des services d'autres modules. Il doit donc envoyer des messages à ces modules qui sous-traitent en partie le service.
- Dresser la liste des services qu'il accepte de rendre à sa prochaine phase d'activité. Il utilise pour cela les primitives *SetServ*, *ResServ* et *PutServ* du noyau NERF. Ces primitives respectivement ouvrent, ferment et positionnent les services (files) de la zone de réception.
- Appeler la primitive *Wait* du noyau NERF, qui termine la phase d'activité.

1.5.2 Phase d'attente

Cette phase d'attente se caractérise par un blocage du module. Au cours d'une phase d'attente le noyau NERF effectue les actions suivantes:

1. Au début de la phase d'attente, le noyau NERF prend en charge l'ensemble des messages de la zone d'émission. L'émission des messages se fera en collaboration avec la station de transport de la couche CORTEX.
2. Le noyau NERF détermine ensuite si le module peut être réveillé. Il scrute pour cela la zone de réception et en particulier les files d'attente dont le service a été validé. La file d'attente est alors ouverte. Deux cas se présentent.

- Il existe des files d'attente ouvertes contenant un message. Le noyau NERF choisit de façon non déterministe une des files ouvertes. Le noyau extrait de la file choisie, le message de tête (gestion FIFO de la file). Le module OMPHALE ne doit pas tenir compte du choix de la file fait par le noyau.
- Les files d'attente ouvertes ne contiennent aucun message. Le module reste en phase d'attente, tant qu'aucun message n'arrive sur une des files ouvertes.

3. Le module est ensuite réveillé. Le message est extrait de la zone de réception ZR. Le noyau NERF retourne au module le numéro du service à activer (la valeur retournée par la primitive *Wait*). Pour accéder au message (recopie dans son contexte), le module OMPHALE utilisera la primitive *LastMessage* du noyau NERF. La primitive *SizeMessage* lui indiquera la taille du message reçu.

1.6 La gestion de la ZE

Au cours d'une phase d'activité, le module place dans sa zone d'émission les messages qu'il désire émettre. Pour cela, le noyau NERF propose trois primitives de gestion de la zone d'émission:

- *CreateMessage*
  Elle ajoute le message passé en paramètre à la ZE. De plus, le noyau NERF vérifie que l'émission est valide c'est à dire que le nom de module est correct. Le mécanisme de contrôle sera détaillé dans la section "protection". La primitive retourne un identificateur de message. Cet identificateur est local au module et pourra éventuellement être utilisé pour une destruction.
- *DeleteMessage*
  Cette primitive détruit un message de la zone d'émission.
- *DeleteAllMessages*
  Cette primitive détruit l'ensemble des messages de la zone d'émission.

La primitive *InitBuffer* est appelée au début de l'exécution du module OMPHALE. En particulier, elle initialise la zone d'émission du module.

1.7 Nommage

Cette section décrit les différents types de nommages dans le système OMPHALE. L'architecture OMPHALE définit des noms pour l'ensemble des entités qu'elle gère.

1.7.1 Noms uniques

L'architecture OMPHALE se doit de définir un mécanisme de nommage des modules. Chaque module va posséder un nom que seul le noyau NERF connaît. Les modules n'ont jamais accès directement aux noms uniques. Ce nom est utilisé par le noyau NERF pour identifier les modules. Le nom unique comprend:

- **Numéro de site**
  Le nom du site de création figure dans le nom unique. A partir d'un nom unique, le noyau détermine immédiatement la localisation du module. OMPHALE de définit pas de mécanisme de migration des modules.
- **Numéro local au site**
  Il permet de distinguer deux modules du même site.

La fonction de création de nouveaux noms uniques est ainsi répartie sur l'ensemble des sites de la configuration. Ceci est possible grâce au champ *numéro de site* du nom unique.

1.7.2 Noms locaux

La désignation de module est réalisée par le mécanisme de **Liens**. Un module peut en désigner un autre au travers d'un **lien** qu'il posséde. Chaque module OMPHALE posséde un ensemble de liens. Ces liens sont stockés dans une structure qu'on appelle l'environnement. Cette structure est gérée par le noyau NERF. Le noyau NERF fournit au module un mécanisme de nommage des liens de son environnement. On appéllera nom local le nom d'un lien dans l'environnement. Si l'environnment est implanté par le noyau NERF à partir d'un tableau, il semble naturel qu'un nom local soit un indice dans le tableau des liens. Quelques remarques:

- La visibilité d'un module se définit à partir de son environnement. Il précise les modules et les services accessibles. La désignation d'un module se fait en utilisant un nom local. La visibilité est représentée par l'ensemble des noms locaux utilisables.
- Si deux modules A et B connaissent un module C, ils posséderont un nom local repérant un lien vers le module C. Un exemplaire du lien vers le module C se trouve dans chacun des deux environnements.

1.7.3 Noms de services

La préparation d'un message OMPHALE (primitive *CreateMessage*) se fait en désignant un service chez le module destinataire. Un nom de service se compose de:

- **numéro de service**: (sélecteur)
  Il permet de désigner un service (une file d'attente) chez le module vers lequel on émet un message.

1.7.4 Noms symboliques

Une application OMPHALE se définit comme un ensemble de modules. A un instant donné, plusieurs applications peuvent cohabiter dans le système. Elle pourront même avoir des modules communs. Le système doit alors de fournir un mécanisme permettant de désigner les modules par des noms symboliques et non plus par des liens. Cette fonctionalité est réalisée par un module (ou plusieurs) du CORTEX qu'on appelle "Le gestionnaire de noms". Ce module définit un nommage symbolique des modules et un ensemble de services sur ce nommage. Ce module gère un catalogue de noms symboliques.

- *Enregistrement*
  Le message qui a servi à l'activation de ce service contient un lien et une chaîne symbolique. Le module GN crée une nouvelle entrée dans son catalogue et y mémorise le nom symbolique et le lien.
- *Recherche*
  Le serveur a reçu un message contenant un nom symbolique. Il recherche dans son catalogue ce nom symbolique. S'il se trouve dans le catalogue, le module GN renvoie ce lien au module client. S'il ne le retrouve pas, le client reçoit un message contenant aucun lien.
- *Destruction*
  Ce service doit détruire une référence symbolique du catalogue.

1.7.5 Noms de messages

La primitive *CreateMessage* place un nouveau message dans la zone d'émission (ZE). La primitive *CreateMessage* renvoie un identificateur de message dans la zone d'émission. Cet identificateur est local au module. Le module peut utiliser cet identificateur pour la destruction du message de la zone d'émission (primitive *DeleteMessage*).

1.8 Protection

La protection est un mécanisme fourni par le système qui contrôle l'accès aux ressources. Les processus OMPHALE sont les modules. Les sections précédentes montrent que les ressources seront perçues en tant que modules. La protection OMPHALE définit un mécanisme de protection qui contrôle l'accès des modules aux autres modules. Le concept de protection est contruit autour de la notion de lien. Les liens (ou capacités) ont fait l'objet de nombreuses études ([FAB 74], [BAS 77], [DEL 79])

1.8.1 Lien

La structure de donnée Lien est locale au noyau NERF. Elle comprend:

- **Un nom unique de module M**
  Le lien définit une liaison uni-directionnelle entre le module qui détient le lien et un module M. Pour réaliser cette liaison, le lien renferme le nom unique du module M. Seul le noyau NERF connaît la structure d'un lien et par conséquence accède au nom unique de module.
- **Les droits sur le module M**
  Pour chaque lien, un champ indique les services accessibles avec ce lien, c'est à dire les méthodes utilisables sur l'objet, ou bien encore les services accessibles sur le module M. Chaque service est utilisable une fois ou éventuellement une infinité. Un compteur d'utilisation est donc associé à chaque service utilisable.
- **La duplication**
  Ce champ précise si la duplication du lien est possible.

Les liens détenus par un module sont stockés dans l'environnement du module.

1.8.2 L'environnement

Pour chaque module, le noyau NERF gère l'ensemble de ses liens. Cet ensemble se trouve dans une structure qu'on appelle l'environnement. Cet ensemble détermine la visibilité d'un module c'est à dire quels sont les modules qu'ils connaît. La désignation d'un module M2 par un module M1 n'est possible que si M1 posséde un lien vers M2.
Cet ensemble de liens peut être implémenté à l'aide d'un tableau. Chaque entrée contient un lien. Certaines entrées pourront être vides. La désignation se fait par indexation dans l'environnement du module émetteur.

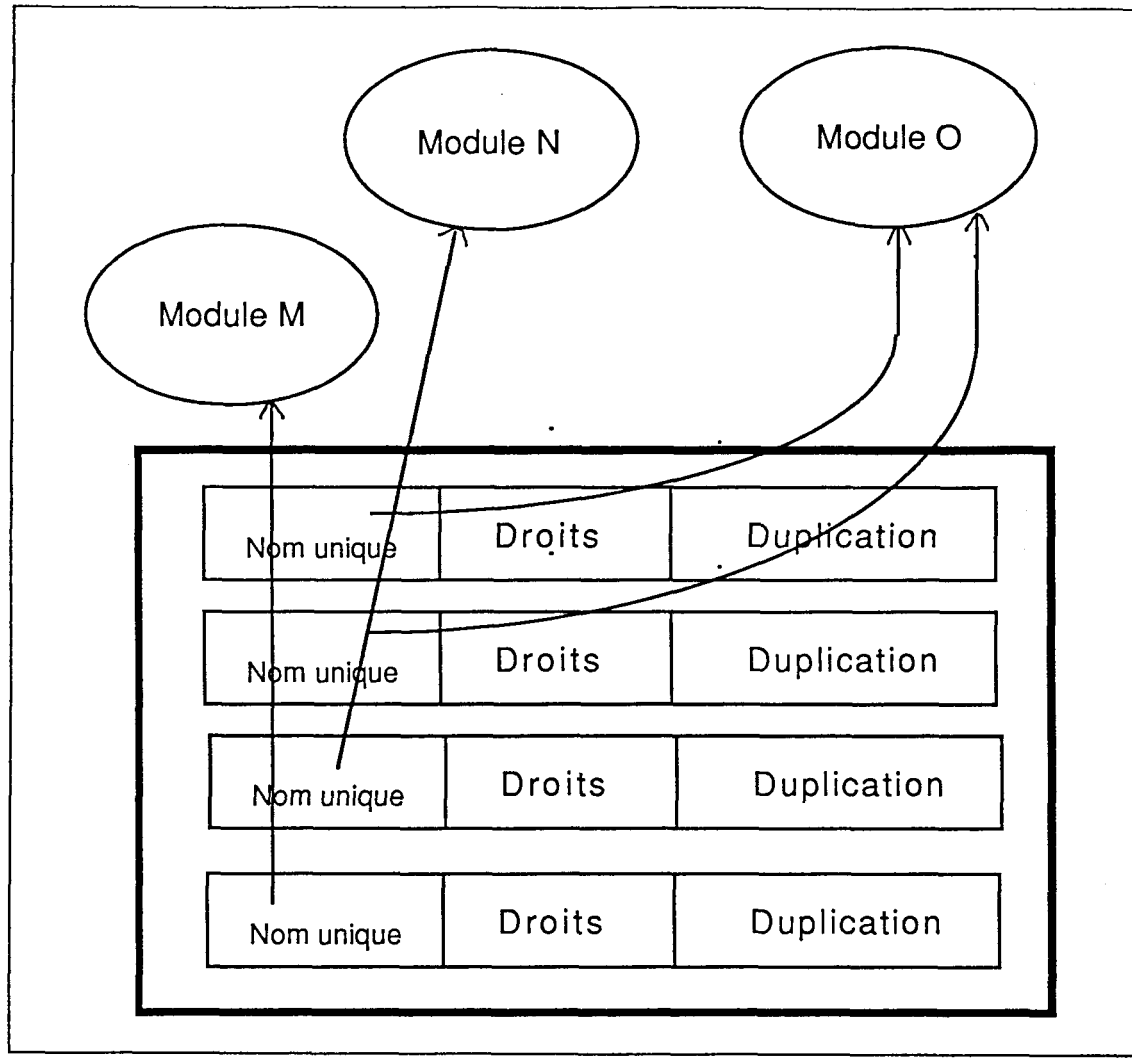


Figure 3. L'environnement

1.8.2.1 Expansion de l'environnement

La visibilité d'un module est étendue par l'ajout de nouveaux liens à l'environnement. Elle se fait de deux manières:

- La réception d'un message contenant des liens entraine l'ajout de ces liens à l'environnement.
- La duplication de liens de l'environnement se fait par la primitive *DuplicateLink*. Cette primitive crée une nouvelle entrée dans l'environnement.

1.8.2.2 Contraction de l'environnement

La réduction de visibilité se fait de trois manières:

- Au moment du transfert de lien dans un message. (primitive *TransferLink*). Les liens sont placés dans le contenu d'un message et quittent donc l'environnement.
- A la destruction explicite de liens de l'environnement par la primitive *DestroyLink*.
- Pour un lien donné, il est possible de restreindre les services accessibles avec ce lien (primitives *SetNoUse*, *SetOneUse*).

1.9 Création et destruction de modules

Cette section décrit les mécanismes de création et destruction de modules. Une grande partie des mécanismes sont externes au noyau NERF.

1.9.1 La création de modules

Quand un module désire en créer un autre, que doit-il faire? Il doit soumettre une requête de création à un module de la couche CORTEX qu'on appelle le gestionnaire de module (GM). Il en existe au moins un par site. Tout module connaît (posséde un lien vers) le gestionnaire de modules de son site. Ce module se comporte typiquement comme un module *serveur*. Il propose des services de création de module.

- *Création locale*
  Ce service consiste à créer un module sur le même site que le module GM. Si le module créateur se soucie de l'éfficacité, il pourra créer sur le même site que lui les modules avec lesquels les communications sont fréquentes. Pour créer un module, le module GM utilise la primitive du noyau *CreerModule* qui crée un module et retourne le lien vers le module crée
  .
- *Création Distante*
  Le message reçu contient une requête de création de module sur un site donné en paramètre. Le module GM possède un lien vers les autres modules GM des autres sites. Cette requête est transmise au module GM du site concerné (*Creation locale*).
- *Création banalisée*
  La création de module aura lieu sur un des sites de la configuration. Le choix de ce site est laissé à l'appréciation du module GM.
- *Création régulante*
  La création du module se fera sur le site le moins chargé de la configuration.

Le noyau OMPHALE ne définit pas de mécanisme de filiation entre les modules. Ceci a pour conséquence que:

- Les exécutions des modules père et fils sont concurrentes.
- Pour se terminer, le module père n'a pas à attendre la fin de l'exécution de ses modules fils.
- Les modules pères et fils ne partagent aucune données. Il n'y a pas de mécanisme d'héritage automatique entre le père et le fils. Le père peut éventuellement **déléguer** une partie de ses droits (liens) à son module fils. Il lui envoie un message d'initialisation contenant les liens dont le fils héritera.

1.9.2 La destruction de modules

A l'instar de la création, la destruction de module OMPHALE est dynamique. La destruction d'un module provient du module lui-même ou sera à l'initiative d'autres modules le connaissant.

- La destruction se fait à l'initiative du module. Deux cas se présentent:
  - **fin normale**
    Le module décide de lui même de s'arrêter. Il invalide l'ensemble des services. Il utilise pour cela *SetServ ({})*. Le noyau NERF s'aperçoit que les services du modules sont tous fermés. Le module meurt.
  - **fin anormale**
    Une errreur de programmation a entrainé un déroutement. Cet événement entraine la fin du module. Les messages de la zone d'émission sont détruits ainsi que les liens se trouvant dans les messages de cette zone d'émission.
- La destruction se fait à l'initiative d'autres modules:
  - La destruction d'un module est automatique si le module n'est plus utilisable. On dira qu'un module n'est plus utilisable, s'il n'est plus connu dans le système (il n'existe plus de liens vers ce module). Le module attend des messages qui ne peuvent arriver. Le module va donc être détruit. Le système gère un mécanisme de ramasse-miettes sur les modules. Cette destruction est automatique sans intervention explicite du programmeur. Les sytèmes par objet nécessitent généralement des ramassee-miettes à cause du grand nombre d'objets.
  - La destruction d'un module peut être demandée par un autre module. Pour chaque module, on définit un service spécifique de destruction. Lorsqu'un message arrive sur ce service, la destruction du module est demandée. Pour détruire un module, il est indispensable de posséder un lien avec le service destruction validé.

A la fin d'un module, les liens de l'environnement sont détruits. La zone de réception est détruite et plus précisément les liens de la zone de réception.

Chapitre 2
Les outils matériels et logiciels

Ce chapitre décrit les outils matériels et logiciels qui ont été utilisés pour l'écriture du noyau de système réparti OMPHALE. Les principaux éléments sont présentés dans ce chapitre. Des compléments peuvent être consultés dans les annexes et les notices techniques du constructeur [BUL86a] [BUL86b] [BUL86c].

2.1 Présentation du SPS7

Le projet SM90 [FIN 81] a été développé par le C.N.E.T. (Centre National d'Etudes des Telecommunications). Ce projet a debouché sur une phase de commercialisation. Les constructeurs BULL (SPS7), TELMAT (SM90) ont proposé des machines type SM90 dans leur catalogue commercial. L'idée maîtresse du projet SM90 était de concevoir une architecture modulaire, multi-processeurs. Cette structure modulaire se justifie particulièrement pour certaines applications où les contraintes suivantes doivent être respectées:

- Possibilité d'ajustement de l'architecture en fonction de l'évolution des besoins de l'application. Augmentation des besoins en temps de calcul, en entrées-sorties...
- Les problèmes de tolérance aux pannes ont fait l'objet de nombreux travaux. Toutes les techniques reposent sur la redondance des matériels et/ou des informations et/ou des traitements. Le projet GOTHIC [BAN 87] illustre parfaitement ce propos. Le système GOTHIC est construit autour du SPS7. En cas de panne d'un module, l'activité est reprise par un autre module. La validité de cette reprise est garantie par l'atomicité des activités et par l'utilisation de mémoire stable.
- Une réponse aux besoins sans cesse croissants en matière de développement.

2.1.1 Le SMBUS

C'est l'outil de communication de l'architecture. Il assure la communication entre les différents modules de l'architecture. Une panne du SMBUS entraine la paralysie totale de la machine.

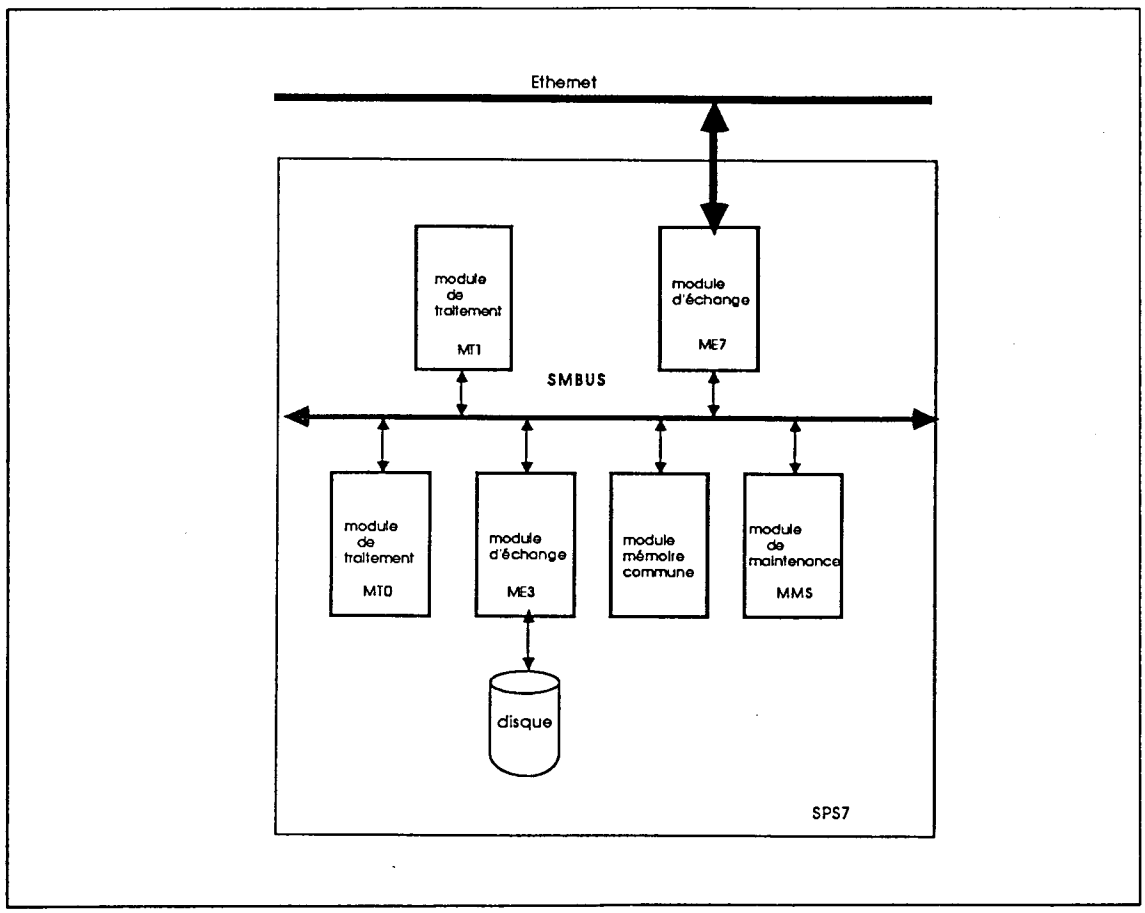


Figure 4. L'architecture

L'architecture est constituée d'un ensemble de modules. Les modules sont connectés au bus de communication (le SMBUS). Ce bus permet la communication et la coopération entre les différents modules. Il se veut suffisamment général pour admettre la connexion de n'importe quel type de module. Le SMBUS définit un protocole de communication qui doit être respecté par les modules. Ce bus comprend:

- Des fils d'adresse (25 fils),
- Des fils de données (16 fils),
- Des signaux:
  - de contrôle de l'intégrité des transferts entre modules.

- de synchronisation et communication inter-modules.
- de diagnostic et de détection de panne.

La question importante est maintenant de savoir comment se fait le partage du SMBUS entre les différents modules.
Les modules sont d'abord répertoriés en deux classes:

- Les modules **esclaves.**
  Ils ont vis-à-vis du SMBUS une attitude passive. En aucun cas, ils n'ont l'initiative d'une requête sur SMBUS.
- Les modules **maîtres**
  Les modules maîtres sont les seuls modules à prendre l'initiative d'un transfert sur le SMBUS. Les conflits d'accès doivent être solutionnés au niveau de ces modules. Un mécanisme matériel d'arbitrage est cablé sur chaque module maître; il utilise la technique du jeton circulant (entre les modules maîtres). La panne d'un ou plusieurs maîtres n'entraine pas le blocage du système.

Il existe un deuxième critère de classification des modules. Ce critère a trait à l'activité du module et non à son attitude par rapport au SMBUS. Avec ce critère, on a alors trois types de modules:

- Les modules de traitement,
- Les modules d'échange,
- Les modules mémoire.

2.1.2 Les modules de traitement

Leur fonction est d'exécuter un système d'exploitation. Ils effectuent des traitements sur les données en mémoire. Les échanges avec les périphériques seront sous-traités par les modules d'échanges. Les modules de traitement sont composés des éléments suivants :

- Un micro-processeur Motorola 680xx [MOT 84] [LEO 86]: Unité de traitement
- Un domaine privé (mémoire privée, temporisateurs, coupleur V24)
- Un domaine local (bus, mémoire locale).

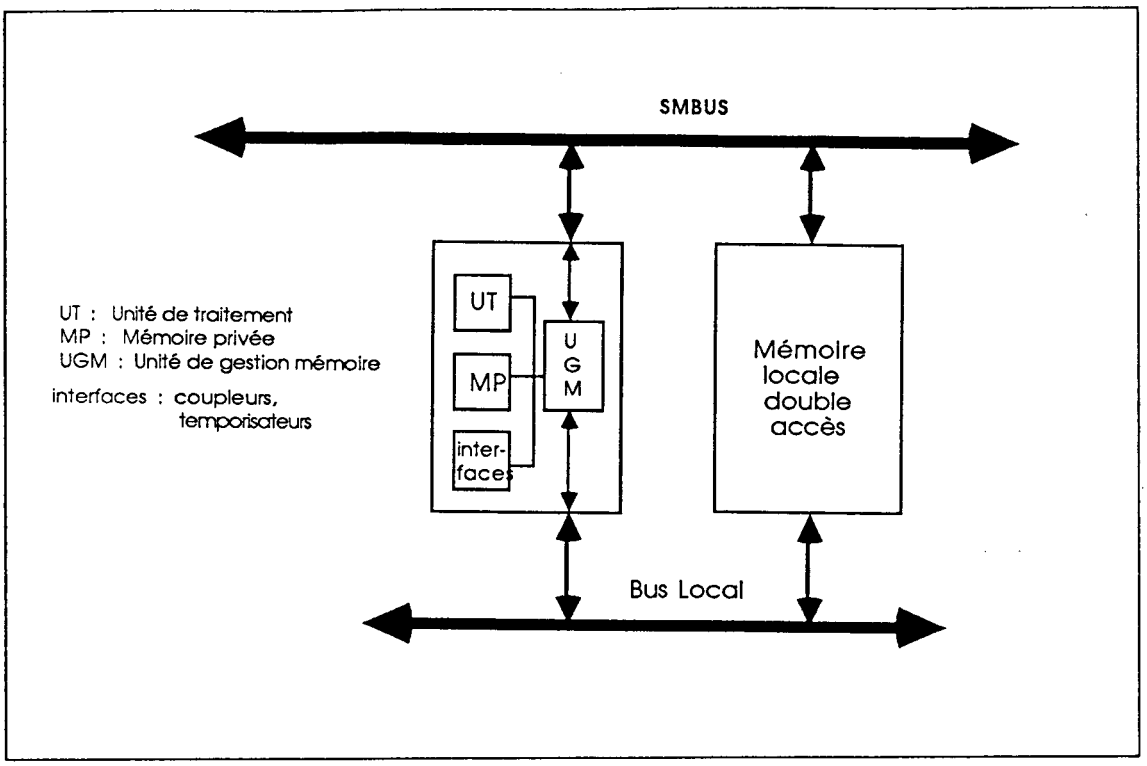


Figure 5. Le module de traitement

Le module de traitement possède une unité de gestion mémoire (UGM). Le processeur manipule des adresses logiques. L'UGM est un dispositif matériel qui assure les fonctions suivantes :

- la conversion des adresses logiques en adresses physiques
  L'UGM est constituée d'une table de descripteurs de segments. A chaque descripteur de segment correspond une adresse physique en mémoire (base du segment). L'adresse logique comprend un numéro de segment et un déplacement. L'UGM traduit donc les adresses logiques en adresses physiques à partir des descripteurs de segment de l'UGM.
- la protection sur les accès aux segments mémoires
  Chaque descripteur de segment de l'UGM posséde des attributs de protection sur le segment. Ces attributs précisent les accés (lecture/écriture) autorisés sur le segment.

L'UGM permet ainsi:

- La protection de l'espace mémoire système contre les accés malveillants des programmes utilisateurs.
- La séparation des espaces d'adressage des processus utilisateurs.

2.1.3 Les modules d'échange

Ils déchargent les modules de traitement de la gestion des périphériques. Ils assurent également le partage de ces périphériques entre les différents modules de traitement. Pour exécuter une opération d'entrée-sortie, le module de traitement soumet sa requête au module d'échange capable de traiter cette requête. La soumission d'une requête se fait par écriture de la requête dans la mémoire d'échange du ME. Le module d'échange analyse et traite la requête. Il pourra éventuellement placer dans la mémoire d'échange le résultat de l'opération (par exemple pour une opération de lecture).

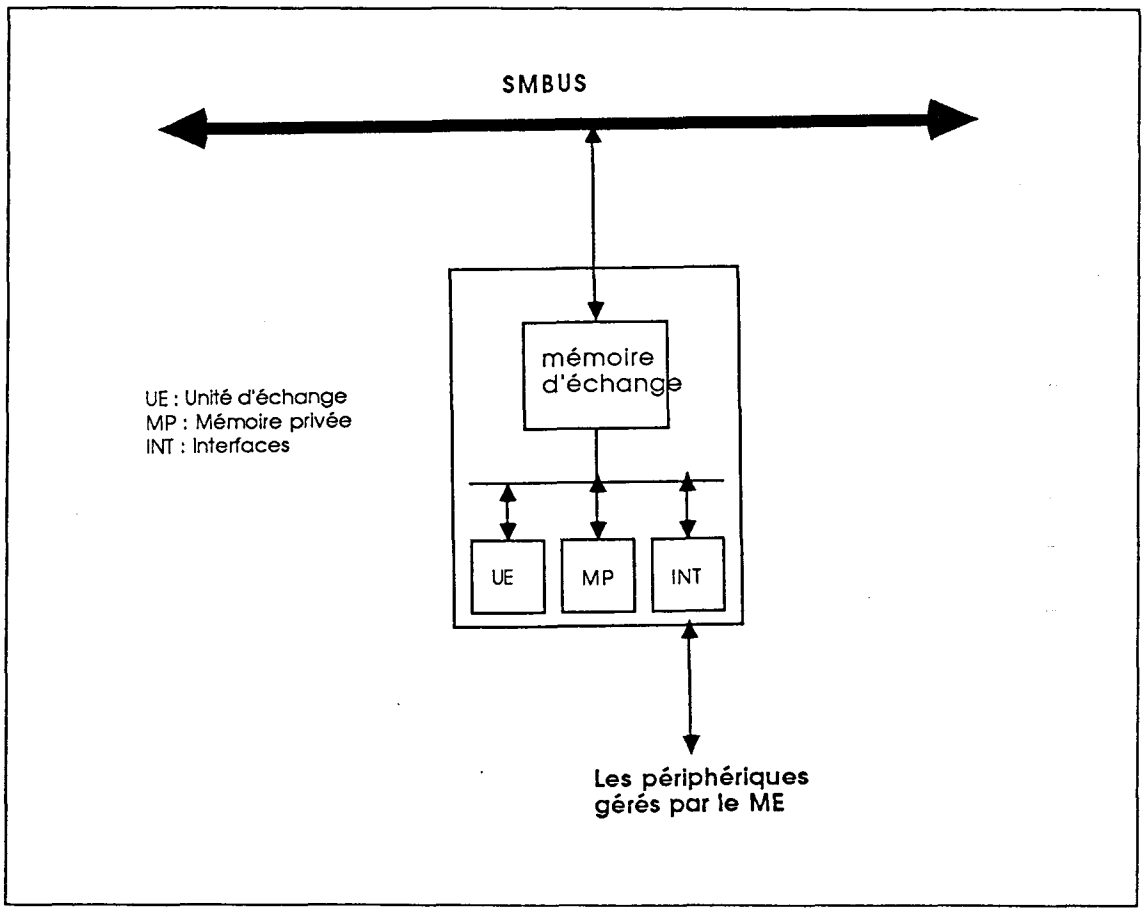


Figure 6. Le module d'échange

Il existe un module d'échange par type de périphérique. On peut citer les modules d'échange suivants:

- Module d'échange ETHERNET [MET 76]
  Il est construit autour d'un microprocesseur 68000. Il permet la gestion des couches réseau propre à ETHERNET (3 premiers niveaux ISO). Il prend en compte les procédures d'émission et réception de trames, gère les trames perdues. Tout accès au réseau transite obligatoirement par le ME ETHERNET.

- Module d'échange disque
  Il effectue le couplage d'équipements à interface SCSI (Small Computer System Interface) conforme à la norme ANSI X 3T9.2. Il prend en compte différents types de contrôleurs permettant le raccordement des disques D570 (56 mégas octets) et de cassettes streamer.

2.1.4 Les modules mémoires

Seul, le module de traitement peut accéder à sa mémoire locale via son bus local. Pour satisfaire les besoins de communication entre les modules de traitement, des modules de mémoires sont connectables au SMBUS. On dira alors que cette mémoire est **globale** ou bien encore **commune**. Toute opération (lecture, écriture) dans cette mémoire transite obligatoirement par le SMBUS. Cette mémoire peut se présenter sous deux formes:

- Une carte mémoire globale connectée directement au SMBUS.
- Des bancs de la mémoire locale du module de traitement pourront être en double-accès c'est à dire accessibles soit par le bus local, soit par le SMBUS.
  Des "switchs" sur la carte permettent de configurer la carte et les bancs mémoire la constituant; pour chaque banc, on indique:
  - Le type d'accés
    Le banc est accessible soit par le bus local, soit par le SMBUS.
  - L'adresse physique du banc.

2.2 Spart : Système temps-rèel

Un système temps-rèel [LAN 87] est capable de supporter l'exécution d'applications où l'environnement impose le respect de contraintes de temps et de débit. Le non-respect de ces dates constitue une défaillance du système. Ce type de système est particulièrement en usage dans les mileux industriels et plus précisement sur le contrôle de machine-outils, les systèmes embarqués.
SPART [ALB 85] [BULL86c] est un noyau temps-réel qui reprend la plupart des concepts définis dans le rapport SCEPTRE [SCE 82] [VOJ 84] du BNI. Devant la grande diversité des noyaux temps-réels, le BNI a proposé une normalisation de ces noyaux. Les idées essentielles de ce rapport étaient les suivantes :

- Cacher au maximum les caractéristiques de la machine.
  Le rapport fait **abstraction** de l'architecture matérielle sous-jacente. Le rapport SCEPTRE définit une machine **abstraite**. Les implantations s'inspirant de ce rapport (SPART en particulier) ont essayé tant bien que mal de respecter ce principe. C'est difficile, car certaines caractéristiques (matérielles) sont difficiles à masquer: mono-processeur ou multi, mémoire locale ou globale.
- Assurer la gestion des ressources de la machine.
  Une des fonctions de base d'un système d'exploitation est d'assurer un partage [CRO 75] [KRA 85] des ressources (matérielles ou logicielles) entre les usagers.
- Garantir la sécurité des opérations du noyau.
  Les applications temps-réel demandent une grande fiabilité. Le noyau et donc ses primitives sont supposées fiables. Aucune anomalie ne peut provenir de l'exécution d'une primitive du noyau.
- Garantir l'atomicité des opérations du noyau.
  L'exécution d'une opération du noyau est indivisible. La réquisition (préemption) du processeur est une opération effectuée par le noyau qui consiste à affecter le processeur à une tâche plus prioritaire. Aucune réquisition du processeur n'est possible pendant l'exécution d'une primitive.
- Permettre l'extensibilité du noyau.
  Pour certaines applications, l'ajout de nouvelles fonctionalités au noyau est impérative pour des raisons d'efficacité et de protection.

Quelques remarques sont à faire:

1. En général, les processeurs possédent au moins deux modes de fonctionnement:
   - **Normal** correspondant à l'exécution d'un programme utilisateur.
   - **Privilégié** (ou mode système) où l'on dispose d'un jeu d'instructions étendu et une visibilité totale de l'espace d'adressage.

   Les opérations du noyau s'exécutent en mode système. Ceci permet d'avoir une visibilité

globale du système et des données s'y rattachant. Mais cela permet également de cacher des informations qui seront donc inaccessibles en mode normal (utilisateur).

2. Les erreurs detectées par les primitives du noyau doivent être signalées au programme d'application. Elles le sont au moyen d'un status renvoyé par la primitive. La non-récupération du code d'erreur par le programmeur se fait sous sa seule reponsabilité. On peut regretter le choix d'une telle solution plutôt qu'un mécanisme d'exception (ADA [DoD 83]).
3. Une interface unifiée multi-langage est fournie autour du noyau. Les primitives du noyau sont utilisables à priori de n'importe quel langage. Pour un langage particulier, il faut y associer les déclarations exactes des objets en termes de types, procédures.

2.2.1 Les objets

L'originalité du rapport SCEPTRE réside dans sa présentation. Le système et le noyau sont entièrement vus en termes de types (ou d'objets). Pour chaque objet, il est défini l'ensemble des opérations attachées à l'objet. La description de SPART [BULL86b] est faite de cette manière. Les objets SPART possédent un attribut supplémentaire:

- **LOCAL** L'objet est crée par un module de traitement et n'est accessible que de ce module. Les ressources nécessaires à l'objet sont allouées dans la mémoire locale du module de traitement.
- **GLOBAL** L'objet est crée par un module de traitement mais est utilisable par n'importe quel autre module de traitement. Les ressources nécessaires à l'objet sont allouées en mémoire globale (commune).

Le noyau SPART de base définit les objets suivants:

- programme,
- tâche,
- segment,
- boîte aux lettres,
- sémaphore,
- délai.

Les objets SPART peuvent être soit GLOBAL soit LOCAL sauf les tâches qui ne peuvent être que LOCAL. Un objet SPART se définit par :

- Un **nom** permettant sa désignation explicite.
- Un **état**, qui est défini à tout instant et peut évoluer au cours du temps.

- Un **ensemble d'opérations** qui permettent notamment:
  - la création ou la destruction d'objet.
  - la consultation ou la modification de l'état d'un objet.
  - la combinaison d'objets entre eux.

L'objet et l'ensemble des opérations s'y rattachant sont définies dans une agence.

2.2.2 L'agence

L'agence est une unité fonctionnelle de construction et de structuration du noyau. Le noyau SPART est ainsi structuré de la manière suivante:

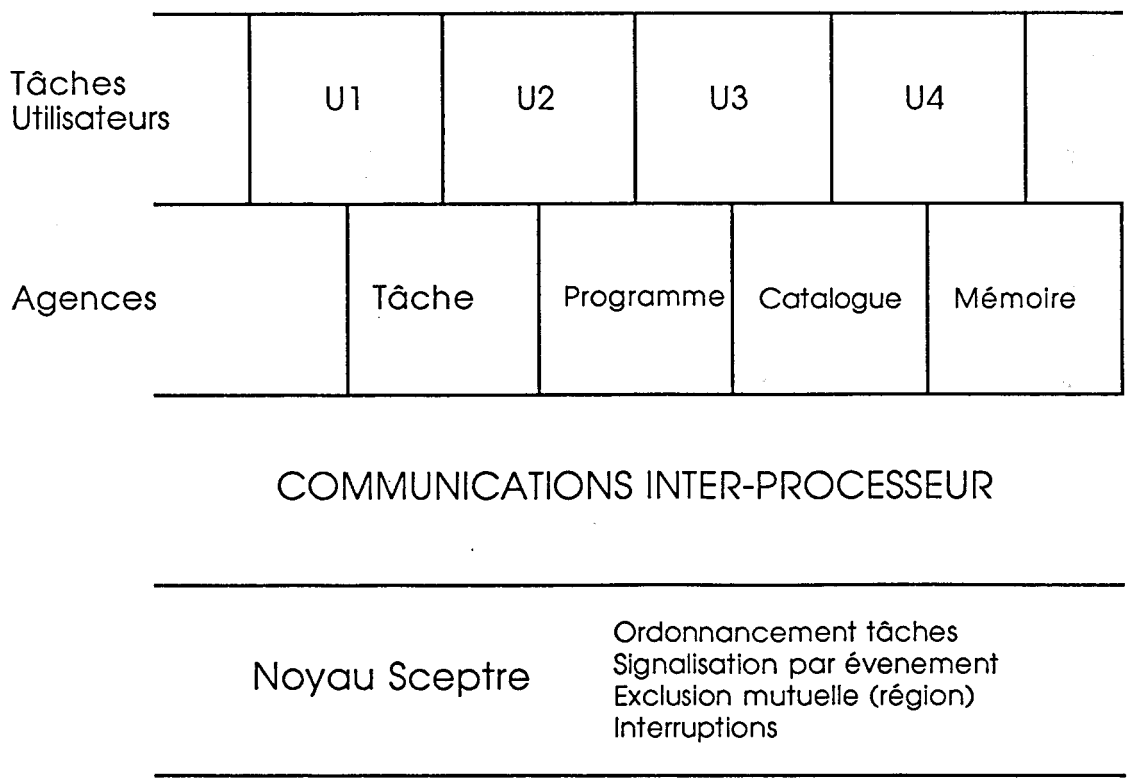


Figure 7. La structuration du système SPART

Une agence définit un type d'objet et l'ensemble des primitives agissant sur cet objet. L'implantation des objets et des primitives est cachée dans l'agence. Pour implanter les primitives, l'agence définira localement des variables, fonctions et même des tâches. Une agence peut utiliser les procédures d'une autre agence.

Les primitives d'une agence sont executées en mode système. L'exécution d'une primitive se déroule de la manière suivante:

1. Contrôler la validité des paramètres de la primitive
   Le contrôle des arguments se fait en mode utilisateur. Le nombre de paramètres, la validité des adresses passées aux primitives est vérifiée. En cas d'erreur dans les paramètres, le traitement de l'erreur a alors lieu en mode utilisateur (et pas en mode système).
2. Placer dans un registre le numéro de la primitive.
3. Passer en mode système.
   A toute agence, on associe un numéro. Pour activer une primitive de l'agence, il suffit alors d'exécuter l'instruction TRAP du 68000 avec le numéro de l'agence.

Le noyau SPART est extensible. L'extension du noyau consiste à intégrer une nouvelle agence. Les principales étapes de l'intégration d'une agence sont:

1. Création du texte source de l'agence. Le texte contient la déclaration du type de l'objet et des primitives (fonctions) d'accès à l'objet.
2. Compilation du texte source de l'agence. Cette phase de compilation aboutit à la création du code objet de l'agence.
3. Edition de lien du nouveau noyau SPART où l'objet de l'agence est inclus. La prise en compte de la modification d'un noyau SPART nécessite l'arrêt du système et une nouvelle mise en route.
4. Modification du contexte d'exécution (run-time) des tâches SPART pour leur permettre d'accéder aux primitives de la nouvelle agence.

2.2.3 Les tâches

La tâche est l'entité active du système qui va correspondre à l'exécution séquentielle d'une suite d'instructions par le processeur. L'exécution d'une instruction ne peut commencer que si la précédente est terminée. C'est un processus séquentiel. Une tâche est locale à un Module de traitement. Une conséquence de ceci est qu'il n'existe pas au niveau du noyau SPART de mécanisme de migration de tâches d'un module de traitement vers un autre.

Comme tout objet SPART, la tâche posséde un nom qui lui est attribué à sa création. A ce nom, le noyau associe un contexte où l'on retrouve les informations suivantes : (voir annexe Contexte d'une tâche)

- registres,

- informations sur l'état de la tâche,
- priorité,
- segment code, pile, etc...

Le noyau SPART accéde aux tâches par l'adresse de leur contexte. La primitive *tache_courante* renvoie un pointeur sur le contexte de la tâche en cours d'exécution. Ce contexte n'est utilisable qu'au niveau du noyau.

```
Contexte *tache_courante()
```

L'agence de gestion de tâches offre les fonctionnalités suivantes:

- Création et activation de tâche,
- Terminaison d'une tâche,
- Attente de la fin d'une tâche,
- Suspension et réactivation de tâche.

Le schéma ci-dessous illustre les différents états d'une tâche SPART.

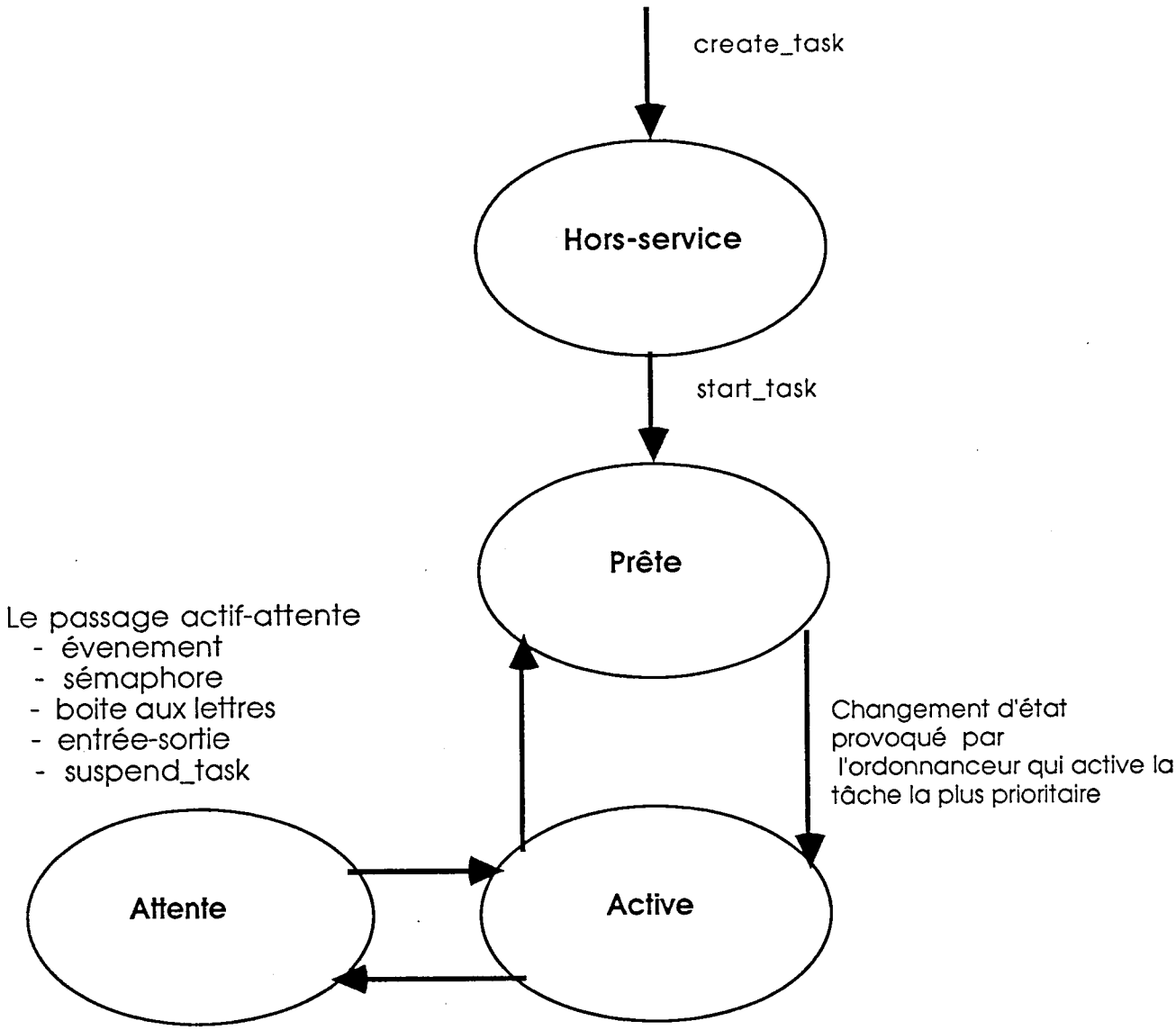


Figure 8. Le graphe des états d'une tâche

- **Hors-service**
  La tâche vient d'être créée mais n'a pas encore été lancée.
- **Prête**
  La tâche est prête à l'exécution mais le processeur est utilisé par une autre tâche.
- **En-cours**
  La tâche est prête et dispose du processeur pour s'exécuter.
- **Attente**
  La tâche se trouve en attente d'un évenement déclenché par une autre tâche tel que l'arrivée d'un message ou par exemple la fin d'une entrée-sortie.

La création de tâches se fait en plusieurs étapes:

1. **Développement croisé sous SPIX**
   Un développement est dit croisé si le résultat a pour cible un système différent du système de développement. La production des exécutables SPART (édition du texte source, compilation et édition de liens) se fait donc sous le système SPIX.

2. **Chargement du code sous SPART**
   Les étapes suivantes sont exécutées sous le système SPART. L'image binaire doit d'abord être chargée en mémoire. La primitive *load* de l'agence programme réalise cette fonction.

   load (numero unite,file_name,prog_type,& prog_id)

   - numéro d'unité désigne le numéro du disque où se trouve le fichier contenant l'image binaire.
   - file_name: nom de fichier SPIX contenant l'image binaire. L'arborescence SPIX est visible sur le système SPART.
   - prog_type: l'image binaire sera réentrante ou pas c'est-à-dire que le code pourra être réutilisé pour d'autres tâches.
   - prog_id: c'est l'identificateur de l'image binaire. Il est nécessaire pour la création de la tâche effective.

3. **Création de la tâche SPART**
   Le chargement de l'image binaire ne suffit pas à créer une tâche. Une tâche n'existe réellement qu'après l'exécution de la primitive *create_task.*

   create_task (prog_id,priorite,privilege,max_resume,& task_id)

   - prog_id: Identificateur de l'image binaire qui est renvoyé par la primitive load.
   - priorité d'exécution de la tâche. (entier de 0 à 31)
   - privilége d'exécution système ou utilisateur.
   - max_resume : Valeur maximum du compteur de réactivation de la tâche.
   - task_id : Identificateur de la tâche créée.

   Au niveau du noyau, la création de tâche consiste à allouer des ressources nécessaires à son execution. Un contexte est alloué pour chaque tâche. Après l'exécution du *create_task*, la tâche est dans l'état hors-service.

4. **Exécution de la tâche SPART**
   A ce niveau de la création, la tâche existe mais son exécution n'a pas encore commencé (Etat hors-service). La tâche sera effectivement active a l'appel de la primitive *start_task.*

   start_task (task_id,message)

   - task_id: nom de la tâche à activer. (issu du create_task).

- message: c'est un message de 8 octets qui sert de paramètre d'initialisation de la tâche. Il est facultatif.

2.2.4 Synchronisation et Communication

Dans un environnement multi-processus, l'accès aux objets partagés peut éventuellement être demandé simultanément par deux processus. L'intégrité et la consistance de ces objets est alors fortement menacée. Les systèmes multi-processus offrent des outils qui gérent ces conflits. SPART définit les outils de synchronisation et communication suivants:

- Les événements,
- Les sémaphores,
- Les boîtes aux lettres.

2.2.4.1 Les événements

L'exécution d'une tâche peut être dirigée par des événements. La poursuite de l'exécution n'est possible que si l'évévenement s'est produit. La tâche peut se mettre en attente d'un ou plusieurs événements. Une tâche peut signaler un événement à une autre tâche.

Il existe deux opérations de base sur les événements:

- Attente
  Une tâche se mettra en attente de l'événement (primitive *wait*).
- Signal
  Une tâche pourra signaler à une autre tâche l'événement (primitive *set_event*)

Un événement est un objet qui peut être dans l'un des deux états "arrivé" ou "non arrivé". Les primitives set_event, reset_event_list, wait permettent d'agir sur ces événements.
On dit que deux tâches sont voisines si elles s'exécutent sur le même module de traitement. La synchronisation par événement n'est possible qu'entre tâches voisines.
L'événement est un outil de synchronisation élémentaire entre tâches. Il n'y a aucune information autre que le signal associée à l'événement.

2.2.4.2 Les sémaphores

C'est un des premiers outils de synchronisation évolué et indépendant du matériel qui ait été proposé ([DIJ 65]). Les sémaphores ont initialement été utilisés pour traiter les problémes d'exclusion mutuelle. L'accés exclusif à des variables d'état est garanti par les sémaphores. Un sémaphore comprend un entier et une file de processus en attente. La primitive P décrémente de 1 la valeur du compteur. Si cette valeur devient négative ou nulle, le processus se bloque. La primitive V incrémente de 1 la valeur

du compteur et relance un processus qui serait bloqué sur ce sémaphore.
De nombreuses variantes des sémaphores ont été proposées tels que par exemple des sémaphores avec messages [SAA 70]. SPART propose une extension intéressante aux sémaphores. Les sémaphores sont souvent utilisés pour contrôler l'accès aux ressources. La primitive *s_request* permet l'accès partagé à la ressource. La primitive *e_request* est utilisé pour l'accès exclusif à la ressource. La programmation de problèmes est parfois simplifiée (par exemple les lecteurs-rédacteurs).

Le reproche que l'on fait souvent aux sémaphores est la difficultés de mise au point des applications utilisant cet outil. Les preuves détaillées des programmes sont souvent indispensables. D'autre part, cet outil n'est pas adapté aux architectures réparties. Il n'est d'ailleurs plus très utilisé dans la conception des systèmes informatiques.

2.2.4.3 Les messages

Dans le cadre d'applications multi-processus, la communication d'informations entre processus est primordiale. Un processus P aura besoin d'une donnée calculée et gérée par un autre processus Q. La communication par messages est particulièrement intéressante, car elle permet à la fois la communication d'informations mais aussi la synchronisation. SPART définit donc l'objet boîte aux lettres. La boîte aux lettres est un objet dans lequel les tâches peuvent placer ou retirer des messages. Chaque boîte aux lettres possède un identificateur qui la distingue des autres. On appelle "capacité d'une boîte aux lettres" le nombre de messages pouvant se trouver dans celle-ci. Cette capacité peut être:

- **nulle**
  La boîte aux lettres ne peut contenir aucun message. L'émetteur d'un message est bloqué tant que le récepteur n'accepte son message. C'est la synchronisation par rendez-vous (ADA [DoD 83], MINIX [TAN 87]).
- **infinie**
  La taille de la boîte aux lettres est virtuellement infinie. Le processus émetteur n'est jamais bloqué. La capacité de cette boîtes aux lettres est en fait limitée aux possibilités matérielles de la machine. A la création d'une boîte aux lettres, la taille de celle-ci n'est donc pas spécifiée.
- **bornée**
  La taille de la boîte aux lettres est finie. Cette taille est spécifiée à la création de la boîte aux lettre. A l'émission d'un message, si le buffer des messages n'est pas plein, le message est déposé dans le buffer de la boîte aux lettres. Si tel n'était pas le cas, la tâche émettrice peut être mise en attente.
  Les boîtes aux lettres SPART sont de capacité bornée définie à la création. L'agence messages définit les primitives suivantes:
  - send (message, mbx_id, delai)
    Envoyer le *message* sur la boîte aux lettres *mbx_id*. Si cette boîte aux lettres est pleine, la tâche est mise en attente pendant le *delai* specifié.
  - receive (message ,mbx_id, delai)

Retirer le *message* de la boîte aux lettres *mbx_id*. Si cette boîte aux lettres est vide, la tâche est mise en attente pendant le *delai* spécifié.

- create_mbx (taille, &mbx_id)
  Une boîte aux lettres de la taille spécifiée est créée. Un identificateur *mbx_id* est renvoyé.

Toute tâche peut communiquer avec d'autres tâches par un nombre quelconque de boîte aux lettres. Deux tâches peuvent communiquer si elles possédent au moins un identificateur commun de boîte aux lettres.

Les boîtes aux lettres peuvent être allouées en mémoire globale. La communication et la synchronisation de tâches s'exécutant sur des MT différents sont ainsi possibles.

L'identité de la tâche émettrice ne figure pas implicitement dans le message. L'envoi de messages est dit **anonyme**. Pour certaines applications, le protocole de communication inter-tâches exige la connaissance de la tâche émettrice. L'émetteur doit alors placer son nom dans le message.

Les boîtes aux lettres SPART sont compatibles avec les boîtes aux lettres UNIX système V. La boîte aux lettres est donc un des outils de communication entre les processus UNIX et les tâches SPART. Pour être accessible d'un processus UNIX, la boîte aux lettres SPART doit être enregistrée dans le catalogue système global (voir la section **catalogue**).

Les messages SPART ne sont pas typés. Pour résoudre cette limitation, le programmeur peut, soit placer dans le message une information sur le type du message, soit créer autant de boîtes aux lettres que de valeurs du type.

Une tâche peut se mettre en attente de message sur une boîte aux lettres. SPART ne définit pas de mécanisme d'attente **multiple** c'est à dire l'attente de message parmi un ensemble de boîtes aux lettres.

Les problèmes classiques de synchronisation peuvent être résolus en utilisant les boîtes aux lettres.

Nous illustrerons ce propos avec le problème du producteur consommateur qui est très représentatif des systèmes d'exploitations. Un processus *producteur* produit de l'information qui doit être consommée par un processus *consommateur*. Par exemple, un pilote (driver) d'imprimante produit des caractères qui sont consommés par une imprimante.

```
                    create_mbx (100, &mbx_info) ;

 /* processus producteur */          /* processus consommateur */

while (1)                           while (1)
 {                                     {
   .                                     .
```

```
    .                                    .
    message = info_produite ;            receive (message,mbxinfo,INFINITE) ;
    send (message,mbxinfo,INFINITE) ;    info_consommee = message
    .                                    .
    .                                    .
}                                    }
```

- La boîte aux lettres *mbxinfo* est partagée entre les producteurs et les consommateurs. Un message de cette boîte aux lettres contient une information produite par le producteur.
- Le processus *producteur* produit lorsqu'il place un message dans *mbxinfo*. Il utilise la primitive *send*. Il sera bloqué si la boîte aux lettres est pleine.
- Le processus *consommateur* consomme en utilisant la primitive *receive* sur *mbxinfo*. Il est bloqué si la boîte aux lettres *mbxinfo* ne contient aucun message.

Contrairement aux sémaphores, la synchronisation par boîtes aux lettres est plus simple à mettre en oeuvre. Il est maintenant admis que la synchronisation par boîtes aux lettres est adaptée aux systèmes répartis. Les boîtes aux lettres SPART n'offrent cependant pas de possibilités de communication entre tâches distantes (s'exécutant sur des machines distantes connectées à un réseau).

2.2.4.4 La gestion mémoire

Trois types de fonctions sont proposés par l'agence mémoire.

- Allocation dynamique de segments
  Un segment est un espace mémoire logique contiguë. Un segment peut être LOCAL ou GLOBAL. Dans ce dernier cas, le segment est accessible par des tâches s'exécutant sur des MT différents. Les primitives d'allocation et libération de segment sont:
  - create_segment (attribute, length, &seg_id)
    Allocation d'un segment de longueur *length* en mémoire locale ou globale (paramètre *attribute*). *seg_id* désigne le segment crée. Le segment n'est pas directement accessible. Pour inclure le segment dans son espace d'adressage, la tâche doit appeler *access_segment.*
  - delete_segment (seg_id)
    Destruction du segment *seg_id.* L'espace mémoire physique est alors libéré. Si le segment était en cours d'utilisation (par d'autres tâches), cette libération ne sera effective qu'à la fin de la dernière utilisation du segment. L'identificateur *seg_id* devient donc invalide.
  - access_segment (seg_id, access_type, &base)
    Accès au segment *seg_id.* Elle effectue la création d'une nouvelle entrée dans la table des segments connus de la tâche. (nouvelle entrée dans l'UGM). *base* désigne le pointeur sur le premier octet du segment. *access_type* précise l'accès demandé sur le segment. (Lecture ou Lecture-ecriture).

- Conversion d'adresse logique en adresse generalisée.
  Pour deux tâches différentes, l'accès au même segment présente une particularité. L'adresse logique n'est pas forcément la même. Pour permettre aux tâches de repérer de façon unique des informations communes, SPART apporte la notion d'adresse généralisée.

  (seg_id,deplacement)

  Les primitives *gen_addr* et *loc_addr* effectuent les conversions d'adresses logiques en adresses généralisées:

  - gen_addr (logical_addr, &general_addr)
    Cette primitive effectue la transformation de l'adresse logique en une adresse généralisée (seg_id,deplacement).
  - loc_addr (general_addr, &logical_addr)
    transformation de l'adresse génerale en adresse logique locale.

- Gestion mémoire dynamique de blocs à l'intérieur d'un segment
  L'allocation est basée sur un découpage du segment en granules. La taille des blocs alloués est arrondie à un nombre entier de granules. L'algorithme gère également l'exclusion mutuelle dans les segments partagés. Deux primitives sont proposées:

  - alloc_memory (seg_id, block_length, &block_address)
    Allocation d'un bloc mémoire dans le segment identifié par *seg_id*. *block_length* indique la taille du bloc à allouer. Cette allocation n'est possible que sur un segment accessible en lecture-écriture. La primitive effectue l'allocation au moyen d'un algorithme "first-fit" (premier bloc de taille suffisante). Elle renvoie l'adresse du début du bloc sélectionné.
  - free_memory (seg_id,block_address,block_length)
    Libération dans le segment *seg_id* du bloc d'adresse *block_address*. *block_length* indique la taille du bloc à libérer. La libération d'un bloc alloué par *alloc_memory* peut se faire par plusieurs appels à *free_memory*. On alloue un bloc mémoire qu'on libère par petits morceaux.

2.2.5 Le catalogue

La création dynamique d'objet a pour conséquence de rendre impossible une résolution statique des identificateurs d'objet. Un mécanisme dynamique de résolution est proposé avec la notion de catalogue. Un catalogue est constitué d'une liste d'association de noms externes (chaîne de caractères) et d'identificateurs d'objets.

Il existe deux types de catalogues:

- Des catalogues d'objets locaux. (objets implantés en mémoire locale). Il en existe donc un par module de traitement.
- Un catalogue des objets globaux. (objets implantés en mémoire globale). Sur une machine donnée, ce catalogue est donc unique.

L'agence catalogue propose trois services:

- catalog (obj_id,extern_name)
  Associer à un identificateur d'objet un nom externe.
  L'identificateur d'objet obj_id est catalogué sous le nom extern_name. Si l'objet est GLOBAL, il est automatiquement catalogué dans la catalogue global, sinon dans la catalogue LOCAL.
- find (attribute,extern_name,out obj_id)
  Rechercher l'identificateur d'objet associé au nom externe.
  attribute détermine le catalogue où s'effectue la recherche (LOCAL ou GLOBAL). Si le nom externe a été trouvé, *id_obj* contiendra l'identificateur d'objet.
- uncatalog (attribute,extern_name)
  Retirer l'association identificateur d'objet, nom externe.
  Destruction d'une référence du catalogue.

2.3 L'utilisation des fonctionnalités SPART

Cette section décrit l'utilisation des fonctionnalités du noyau SPART. Certains exemples sont d'ailleurs utilisés dans le chapitre implantation du noyau NERF.

2.3.1 La gestion de listes

La structure de données **liste** et en particulier **file** est souvent manipulée par le noyau des systèmes d'exploitation. On peut citer:

- Les files de descripteurs de processus. Les noyaux construits autour des moniteurs utilisent souvent cette structure. Une file de processus en attente est associée à chaque condition. Ces files sont alors gérées par priorité croissante de processus.
- Les algorithmes de gestion mémoire, pagination, fichiers sont construits autour des files. La liste des blocs libres d'un disque ou bien alors la liste des blocs libres d'un segment.
- Les messages en attente de traitement sur une boîte aux lettres sont gérées en FIFO.
- Un serveur d'impression (spooler) gère les demandes d'impression de fichier à l'aide d'une file FIFO.

Dans notre noyau, les listes seront utilisées pour les objets suivants.

- La zone de réception du module OMPHALE contient les messages qui n'ont pas encore été traités. Pour chaque service du module, les messages en attente sur ce service sont mémorisées dans une liste gérée en FIFO.

- La zone d'émission du module (messages OMPHALE) contient les messages émis par le module au cours de sa dernière phase d'activité. Elle est représentée sous la forme d'une liste chaînée de messages.

On a vu que pour deux tâches SPART distinctes, les adresses logiques d'un même emplacement mémoire pouvaient être différentes. Pour pouvoir partager des listes entre plusieurs tâches, il est impératif de manipuler des adresses relatives (déplacements par rapport au début du segment). Ces tâches vont constamment utiliser des fonctions de conversions d'adresses absolues en adresses relatives (déplacement par rapport au début du segment).

```
unsigned int AddAbsol (AddRel)
unsigned int AddRel ;
{
 return (AddRel?BaseSyst+AddRel:NIL) ;
}
```

La fonction AddAbsol convertit une adresse relative en adresse absolue.

- *AddRel*: Adresse relative à convertir.
- *BaseSyst*: Pointeur absolu sur le début du segment.
- Lorsque le déplacement vaut *NIL* (0), le pointeur NIL est retourné.

La fonction AddRelat convertit une adresse absolue en adresse relative.

```
unsigned int AddRelat (AddAbsol)
unsigned int AddAbsol ;
{
 return (AddAbsol?AddAbsol-BaseSyst:NIL) ;
}
```

- *AddRel*: Déplacement à convertir en adresse absolue.
- *BaseSyst*: Adresse absolue sur le début du segment.

2.3.2 Gestion mémoire

Au niveau de l'agence mémoire, le noyau SPART offre une grande richesse d'outils pour la gestion de la mémoire centrale. Cette pléthore d'outils peut rendre la programmation difficile à la lecture. D'où la nécessité de créer des outils simples à utiliser et efficaces. Souvent, le programmeur doit gérer sa mémoire. Il utilise pour cela des procédures d'allocation et libération mémoire. Il doit donc retrouver deux types de primitives:

- new, malloc
  Allocation d'un emplacement mémoire d'une taille donnée en paramétre. La primitive renvoie un pointeur sur la zone allouée.
- dispose, free
  Restitution d'une zone mémoire passée en paramètre ainsi que la longueur de cette zone.

La fonction *NewGlobal* alloue une zone mémoire de la *taille* indiquée en paramétre dans le segment système *SegIdSyst.*

```
unsigned int NewGlobal (Taille)
int Taille ;
{
 unsigned int Pt=0 ;
 status cr ;

 cr=alloc_memory (SegIdSyst,Taille,&Pt) ;
 if (cr == OK)
   return (Pt+BaseSyst) ;
   else return NIL ;
}
```

Il est peut être utile de faire quelques commentaires sur cette fonction:

- *SegIdSyst*: Identificateur du segment système où l'allocation sera faite.
- *BaseSyst*: Pointeur sur le debut du segment système.
- *alloc_memory* retourne un déplacement dans le segment sur la zone allouée. Il faut donc ajouter l'adresse de début au déplacement pour obtenir un pointeur absolu sur la zone allouée.
- Si le segment est saturé (il n'existe plus de bloc libre à la taille spécifiée), la fonction renvoie le pointeur NIL.

La procédure *DisposeGlobal* assure la libération de l'espace mémoire dont l'adresse et la taille sont passées en paramétre.

```
void DisposeGlobal (Pt,Taille)
int Taille ;
unsigned int Pt ;
{
 free_memory (SegIdSyst,Pt-BaseSyst,Taille) ;
}
```

- *Pt*: Pointeur sur la zone mémoire à libérer.
- *Taille*: Taille de la zone mémoire à libérer.
- *SegIdSyst*: Identificateur du segment système.
- *BaseSyst*: Pointeur sur le début du segment système.
- La primitive *free_memory* demande un déplacement pour l'emplacement à libérer. Il faut donc calculer ce déplacement.

2.3.3 Les messages de longueur variable

Les messages de taille variable demandent une gestion mémoire au niveau du noyau plus complexe. Mais le travail des programmeurs est simplifié. La taille des messages SPART est limitée à 8 octets (message de taille fixe). Cette restriction rend le travail de programmation plus lourd. Par exemple, le transfert de message de taille plus importante pose des problèmes. Plusieurs solutions sont envisageables:

- Multiplier le nombre de messages en envoyant autant de messages que nécessaires.
- La solution précédente présente l'inconvénient de la multitude des messages. La solution que nous proposons nécessite un segment mémoire qui contient les messages à transmettre. A chaque envoi correspond un message SPART qui contient une adresse (relative ou déplacement) dans le segment mémoire. L'envoi d'un message de taille variable consiste donc à:

```
char MessSpart [MESS_LENGTH] ;
xx_id Mbx;
xx_id SegIdMessage ;
unsigned int Deplacement ;
char * Message ;
int tab [100] ;

    .
    .
    . /* Initialisation du tableau tab par la
    .    tache emettrice                    */
    .
/* transfert du contenu du tableau tab dans un message */

Message = NewGlobal (sizeof (tab)) ;

memcpy (Message,tab,sizeof (tab)) ;

Deplacement = AddRelat (Message) ;

memcpy (MessSpart,&Deplacement,sizeof (unsigned int)) ;
send (MessSpart,Mbx,INFINITE) ;
```

  - Allouer un emplacement dans le segment des messages.
  - Recopier dans cet emplacement le contenu du message.
  - Calculer le deplacement par rapport au début du segment.
  - Copier dans le message le déplacement.
  - Emettre le message SPART vers le destinataire.

Les opérations duales sont exécutées à la reception du message.

```
char MessSpart [MESS_LENGTH] ;
```

```
char *Message ;
xx_id Mbx;
xx_id SegIdMessage ;
unsigned int Deplacement ;
int tab [100] ;

receive (MessSpart,Mbx,INFINITE) ;

memcpy (&Deplacement, MessSpart,sizeof (unsigned int)) ;

Message = AddAbsol (Deplacement) ;

memcpy (tab,Message,sizeof(tab)) ;

DisposeGlobal (Message,sizeof(tab)) ;
    .
    . Utilisation du tableau tab par la
    . tache receptrice
    .
```

– Lecture de la boîte aux lettres.

– Retrouver le déplacement dans le message SPART et recalculer l'adresse absolue.

– Lecture du contenu du message dans le segment des messages.

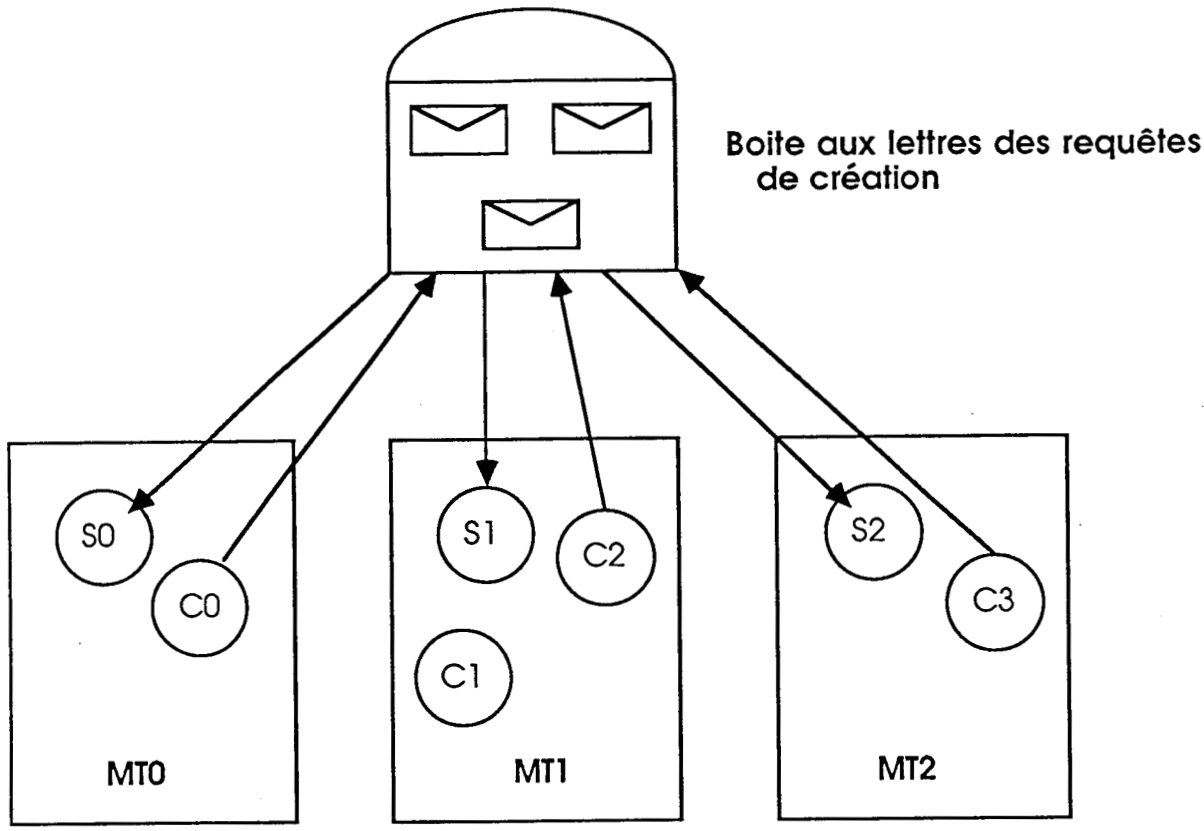
– Libérer l'emplacement alloué dans le segment des messages.

2.3.4 La création de tâche distante

A la section précédente nous avons vu que l'agence tâche proposait une primitive de création de tâche (*create_task*). Cette primitive ne permet pas de créer des tâches sur d'autres modules de traitement que celui sur lequel cette primitive est exécutée. Comment contourner cette contrainte? La solution retenue a été d'activer sur chaque module de traitement une tâche dont la fonction principale est d'activer des tâches sur leur module de traitement. Ce sont des tâches "**serveur**". Les requêtes aux tâches "**serveur**" se trouvent dans une boîte aux lettres. Cette boîte aux lettres se trouve en mémoire globale pour que toutes tâches du système puissent y accéder. Une requête de création de tâche se trouve dans un message qui a été émis par une tâche "cliente". Ce message contient les informations suivantes:

- nom du fichier contenant le programme exécutable.
- programme réentrant ou pas.
- priorité de la tâche à créer.
- message initial de la tâche.
- l'identificateur de la boîte aux lettres de réponse.

La taille d'une telle requête étant superieure à 8 octets. Il faut donc utiliser la technique décrite dans la section précedente.



Si: Tâches serveuses de création de tâches
Lecture Boite aux lettres

Ci: Tâches clientes demandant des créations.
Ecriture Boite aux lettres.

Figure 9. La création de tâche distante

Chapitre 3
Une implantation du noyau NERF

3.1 Le site OMPHALE

Le système a été développé sur les SPS7 du LIFL (Laboratoire d'Informatique Fondamentale de Lille).

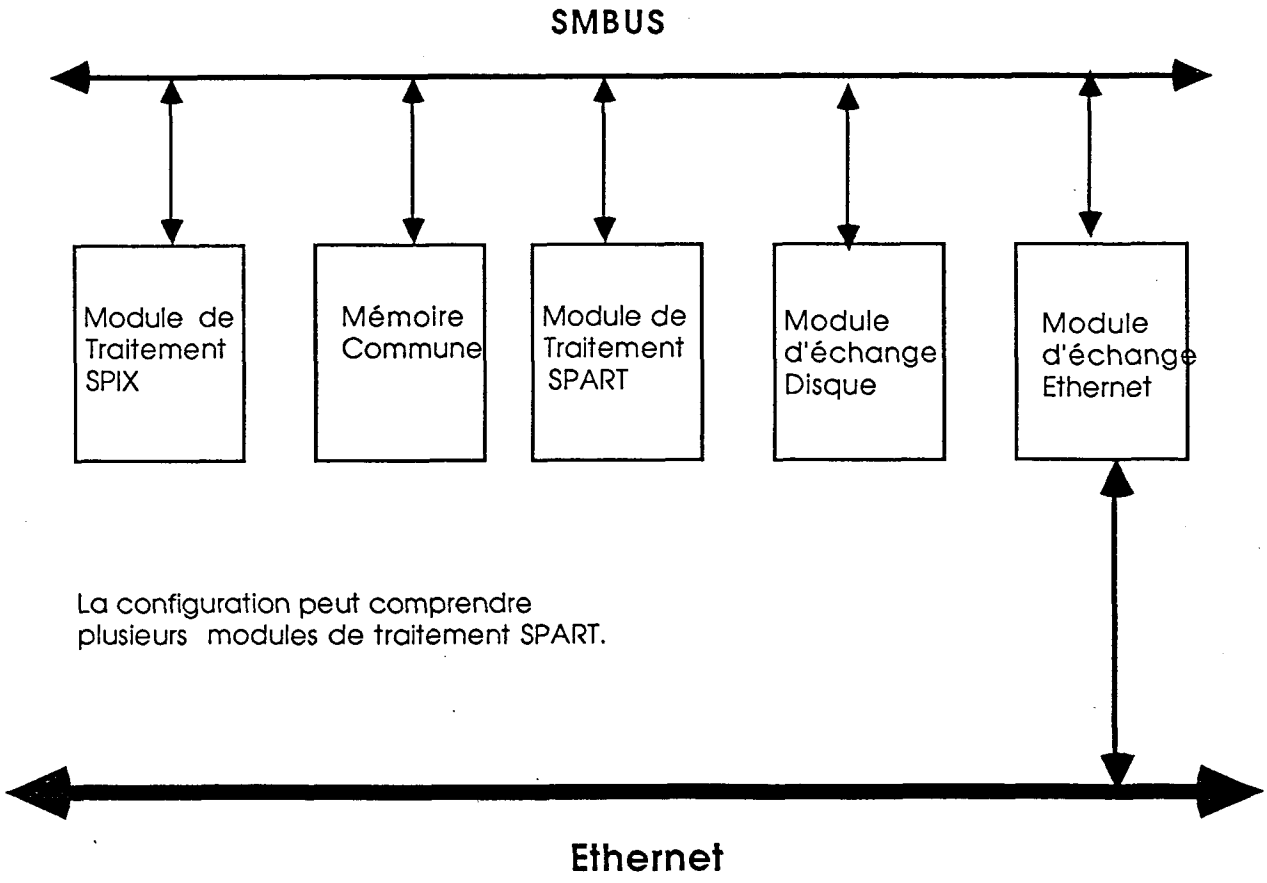


Figure 10. La configuration matérielle

La configuration d'une SPS7 est la suivante:

- Un module de traitement fonctionnant sous UNIX (MTU). Deux mégas octets de mémoire locale sont connectés au bus local. Les principales fonctions du MTU sont:
  - Le développement des applications OMPHALE. Un programmeur OMPHALE utilisera la richesse des systèmes UNIX [RIT 74] et en particulier :

- Les outils de développement de programme tels que les éditeurs de textes, compilateurs, assembleurs et éditeurs de liens.
- La documentation consultable en ligne.
- Les outils de production de documents.

– La station de transport est bâtie autour des sockets TCP/IP [SUN 86]. Sa fonction principale est de transférer les messages entre modules distants (émetteur et destinataire du message ne réside pas sur le même site).

– La gestion des sites OMPHALE connectés assurée par des processus UNIX.

- Au moins un module de traitement sous SPART (MTS). Les Modules de traitement SPART supportent l'exécution du noyau du système OMPHALE, la couche CORTEX, ainsi que les applications.
- Un méga octets de mémoire globale. La mémoire globale permet la communication entre:
  – MTU et MTS. En effet les zones d'émission sont partagées avec la station de transport.
  – Les différents MTS. Si la configuration est composée de plusieurs modules de traitement SPART, la communication entre les MTS se fait par la mémoire globale.
- Un module d'échange disque et un (voire plusieurs) disques. L'accès au disque se fait par l'intermédiaire d'un module d'échange. L'image du noyau NERF réside sur un des disques du site. L'image du noyau est chargée au démarrage de la machine. De plus, le disque conserve les images binaires des modules.
- Un module d'échange ETHERNET. La communication avec les autres sites du système est à la charge de la station de transport via ce module d'échange. Le réseau ETHERNET est accessible par le module d'échange ETHERNET.

Au niveau logiciel, le noyau NERF est constitué du noyau temps-réel SPART auquel on a intégré trois nouvelles agences:

- Messages Omphale.
- Liens.
- Routage.

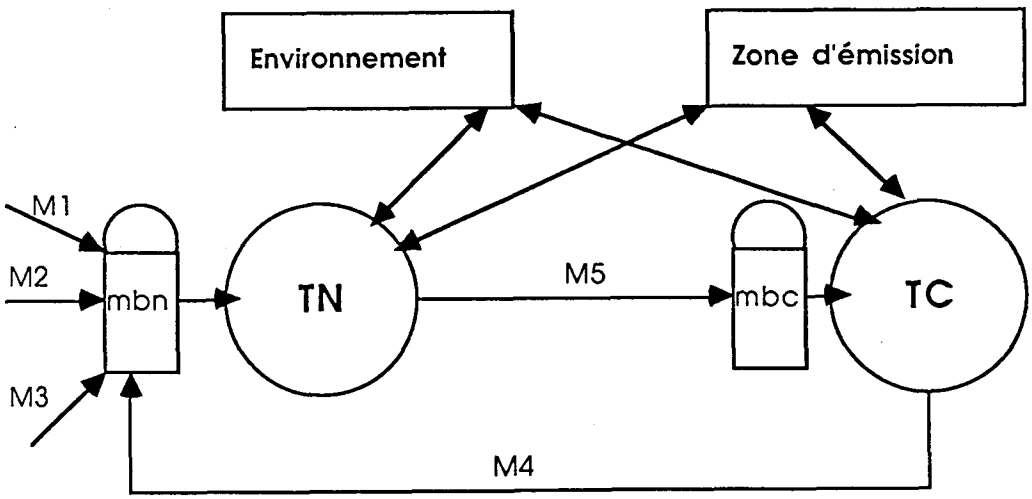
Les objets et les primitives définies par ces agences sont décrites dans les sections suivantes.

3.2 Le module OMPHALE

Le système OMPHALE supporte l'exécution de **modules**. Cette section décrit la réalisation du module OMPHALE sur un site.

3.2.1 Implantation d'un module OMPHALE

Un module OMPHALE peut se définir comme un couple de tâches s'exécutant en parallèle. On appelle TN la tâche noyau et TC la tâche calcul. MBN et MBC sont des boîtes aux lettres nécessaires pour la communication entre TN et TC. De plus, trois structures de données (Zone d'émission, Zone de réception et environnement) sont associées à chaque module.



M1 Duplication de référence
M2 Destruction de référence
M3 Message externe
M4 Message d'attente
M5 Message de réveil.

Figure 11. Le module

1. La tâche TC exécute les calculs utilisateurs c'est à dire le code des services et le contrôleur du module. Les appels systèmes OMPHALE se font au travers des agences Messages et Liens. Durant une phase d'activité, la tâche TC gère sa zone d'émission ZE et son environnement.
2. La tâche TN gère la zone de réception et l'environnement du module. Les fonctions de routage des messages de la ZE sont exécutées par TN en utilisant les primitives de l'agence Routage.

3. La boîte aux lettres MBC sert à la communication entre TN et TC. Elle contient au plus un message SPART émis par TN. C'est un message de réveil de TN à destination de TC. (voir ci-après).
4. La boîte aux lettres MBN contient des messages systèmes (SPART). Ils sont lus par la tâche TN. Quatre types de messages peuvent se trouver dans cette boîte aux lettres:
   - Un message SPART d'attente de TC
     Le module OMPHALE s'est mis en attente. La tâche TC a exécuté la primitive *Wait*. Cette primitive émet un message à destination de TN pour lui signifier la fin de la phase d'activité.
   - Un message SPART externe
     Le module OMPHALE a reçu un message d'un autre module. La tâche TN récupère ce message et le place dans la zone de réception du module.
   - Un message SPART de destruction
     La tâche TN est prévenue qu'une destruction de lien vers le module a été effectuée. Le compteur de duplication de référence doit être décrémenté. L'étude sera complétement détaillée dans la section "agence Routage".
   - Un message SPART de duplication
     Une duplication d'un lien vers le module a été effectuée. Le compteur de duplication doit être incrémenté.
5. La **zone d'émission** (ZE) contient les messages OMPHALE émis par le module durant la dernière phase d'activité. Pour permettre la communication avec la station de transport (sous UNIX), la zone d'émission est implantée dans la mémoire système globale. A la fin de la phase d'activité du module (après le *Wait*), les messages de la ZE sont émis vers les modules destinataires. Cette émission est à la charge de la tâche TN du module et du système de transport des messages.
6. La **zone de réception** (ZR) est gérée par la tâche TN. Elle contient les messages reçus par le module et non encore traités par celui-ci. Une file d'attente (quai) est structurée comme une liste chaînée de messages. La ZR est structurée comme un tableau de liste de messages. La zone de réception est localisée dans un segment système local à la tâche TN.
7. L'**environnement** du module contient l'ensemble des réferences possibles vers les autres modules du système.

3.2.1.1 Remarques

Une des originalités de l'implantation du module OMPHALE, est la réalisation de celui-ci à partir d'une couple de deux tâches SPART: Cette solution a les avantages suivants:

Structuration du noyau
Chaque tâche TN ne gère que les structures de données du module. On évite ainsi le développement d'un noyau manipulant la totalité des structures des modules du site. L'activité du noyau NERF est ainsi représentée par les différentes activités des tâches noyau TN.

Parallélisme d'exécution du module
Le découpage de l'exécution du module OMPHALE en deux tâches parallèle exploite les possibilités de parallélisme de l'architecture. Dans le cadre d'une configuration avec deux modules de traitement, on peut ainsi spécialiser l'activité des modules de traitement. Un module de traitement supporte l'exécution des tâches noyau et l'autre module de traitement celles des tâches calcul. L'étude de l'architecture spécialisée du site 0 (à changement de contexte rapide) se révéle pleine d'enseignements.

3.2.1.2 Cheminement d'un message OMPHALE

Le système OMPHALE propose un outil de communication asynchrone d'envoi de messages entre les modules. Ce paragraphe décrit le cheminement complet d'un message OMPHALE allant de son émission jusqu'à sa réception.

1. Au cours d'une phase d'activité, un module peut préparer des messages OMPHALE. La préparation du message se fait au moyen de la primitive *CreateMessage*. La primitive *CreateMessage* place le message OMPHALE dans la zone d'émission du module.
2. La fin de la phase d'activité du module OMPHALE se caractérise par l'appel de la primitive *Wait* par la tâche TC. La primitive *Wait* se passe en deux temps:
   - Un message SPART d'attente est émis vers la boîte aux lettres MBN pour prévenir TN que TC est en attente.
   - La tâche TC se met ensuite en attente d'un réveil provenant de TN (message dans la boîte aux lettres MBC)

   Dès qu'un message SPART de réveil arrive à TC, une nouvelle phase d'activité commence.
3. TN reçoit le message SPART provenant de TC. Les messages OMPHALE de la zone d'émission ZE vont être pris en charge par TN. Pour chaque message de la zone d'émission deux cas se présentent:
   - Si le message est à destination d'un module local (résidant sur le même site), un message SPART externe est émis vers la boîte aux lettres MBN du module destinataire.
   - Si le message est à destination d'un module distant (ne résidant pas sur le même site), un message SPART est émis vers le système de transport qui va assurer l'émission du message au travers du réseau.
4. Le message SPART est reçu par la tâche TN du module destinataire. Ce message SPART contient un pointeur vers le message OMPHALE. Cette tâche TN place le pointeur sur le message OMPHALE dans la zone de réception du module.
5. Si le module était en attente et que le message arrive sur un quai ouvert, TN réveille alors la tâche TC par émission d'un message SPART. C'est la fin de la primitive Wait. Les liens du message OMPHALE choisi sont ajoutés à l'environnement du module.
6. La primitive *LastMessage* copie le contenu du message OMPHALE dans le contexte de la tâche TC.

3.2.2 Etats du module

Un module OMPHALE se trouve dans l'un des états suivants.

- **Actif**
  Le module est dans une phase d'activité. La tâche TC est alors active. Pendant ce temps, la tâche TN peut recevoir les messages provenant des autres modules et les placer dans la zone de réception du module.
  Une phase d'activité se termine par l'exécution de la primitive *Wait* (par la tâche TC).

- **Attente**
  Le module s'est mis en attente par l'exécution de la primitive *Wait*. La tâche TC est bloquée. La tâche TN est réveillée et sera responsable du réveil de la tâche TC. TN réveille TC si il existe un quai ouvert de la zone de réception possédant un message. TN est ensuite activée à chaque réception de message OMPHALE.

- **Mourant**
  L'état mourant du module précéde la mort. Cet état est irréversible. Il conduira irrémediablement à la mort du module. Le passage à l'état **Mourant** marque la mort de la tâche TC et la destruction de la boîte aux lettres MBC. Il n'y a aucune possibilité de revenir à l'état **Attente** ou **Actif**. Le passage de l'état **Attente** à l'état **Mourant** peut résulter de plusieurs situations:

  – Le module n'est plus réactivable. Le noyau (la tâche TN) s'aperçoit que la totalité des quais du module sont fermés. Le module ne repassera jamais à l'état **actif**.

  – Le module n'est plus référençable dans le système. Il n'existe plus aucun module dans le système possédant un lien vers ce module. Le module n'est donc plus connu excepté par lui-même (lien 0).

3.3 Les structures de données du noyau

3.3.1 Nommage interne des modules

Tout système informatique définit un mécanisme de désignation des objets qu'il gére. Chaque objet posséde un nom unique permettant sa désignation. Les objets gérés par le noyau OMPHALE sont les modules. Nous proposons le mécanisme de désignation interne où le nom du module est composé de deux champs:

- L'adresse IP du site de création du module. En effet, le réseau local Ethernet fonctionne sous TCP/IP. L'adresse Internet IP tient sur 32 bits. Elle spécifie un numéro de réseau, et le numéro de l'ordinateur sur ce réseau. Cette adresse IP est donc unique.
- L'identificateur de sa boîte aux lettres SPART MBN. A un instant donné, un identificateur d'objet SPART désigne un objet et un seul. Ce qui garantit l'unicité. Les identificateurs d'objet SPART sont réutilisables, c'est à dire qu'après la destruction d'un objet, un nouvel objet créé pourra posséder le même identificateur d'objet. Les noms uniques de modules OMPHALE sont donc réutilisables.

3.3.2 Les structures gérées par l'agence Liens

3.3.2.1 Lien

C'est l'entité de désignation et de protection du système. Pour garantir la protection, les modules n'accèdent à cette structure que par les primitives de l'agence; Ils ne peuvent donc pas manipuler directement la structure de donnée *Lien*. Le type *Lien* est défini comme suit:

```
typedef
    struct
          {
           unsigned int SiteName  ;
           xx_id    Destinataire ;

           unsigned short UseOrNo:MAXSERVICES ;
           unsigned short OneOrNUse:MAXSERVICES ;

           char MayDuplicate ;
          } Lien ;
```

La structure Lien se compose des champs suivants:

- Le nom unique du module désigné par le lien. Il est constitué des composants suivants:

- *SiteName*: numéro INTERNET du site d'exécution du module.
- *Destinataire*: numéro de la boîte aux lettres MBN du module.

- *UseOrNo* donne la liste des services accessibles avec ce lien. Si le i^{eme} bit de *UseOrNo* est à 1, le i^{eme} service est utilisable. Si il est à 0, ce service n'est pas accessible.
- *OneOrNUse* précise le nombre d'utilisations possibles du service. Si le i^{eme} bit de *OneOrNUse* est à 1, le service est accessible autant de fois que nécessaire. Si il est à zéro, le service n'est accessible qu'une seule fois.
- *MayDuplicate* précise si le lien est duplicable ou pas. Si *MayDuplicate* vaut TRUE, le lien peut être dupliqué. Si ce champ est à FALSE, le lien n'est pas duplicable.

3.3.2.2 L'environnement

L'ensemble des liens détenus par le module est stocké dans son environnement. Tout module possède un environnement qu'il est le seul à manipuler par l'intermédiaire des primitives de l'agence Liens. Un module détient au plus 32 liens. Le tableau a été choisi comme structure de données pour l'environnement. Le module ne manipule pas directement les liens mais des identificateurs de liens. Ces identificateurs sont locaux à un module. Le type *Environnement* est défini comme suit:

```
typedef struct {
                char EtatEntree ;
                unsigned short int Version ;
                Lien LeLien ;
               } Env [SizeOfEnvironnement] ;
```

- La taille de l'environnement est fixé par la constante *SizeOfEnvironnement* définie dans le fichier ConstNerf.h. Elle est définie actuellement à 32.
- Le champ *EtatEntree* précise l'état de l'entrée de l'environnement. Ce champ peut avoir l'une des valeurs suivantes.
  - *Vide*
    Cette entrée ne contient aucun lien. Cette entrée est utilisable pour placer un nouveau lien dans l'environnement (La primitive *DuplicateLink* ou à la réception d'un lien se trouvant dans un message).
  - *LienPresent*
    Cette entrée contient un lien. Ce lien a été crée par la primitive *DuplicateLink* ou à la réception d'un message avec lien
  - *LienPermanent*
    Cette entrée contient un lien qui est permanent. Ce sont les entrées 0, 1 et 2 qui désignent respectivement le module lui-même, le gestionnaire de noms et le gestionnaire de modules. Un lien Permanent n'est pas destructible (primitive *DestroyLink*).
  - *LienMessage*
    Le lien qui se trouvait dans cette entrée a été transféré dans un message (au moyen de la primitive *TransferLink*).
  - *LienADetruire*
    La destruction du lien a été demandée par la primitive *DestroyLink*.
- Le champ *Version* désigne un entier. Un nom local de lien est la concaténation du numéro de *Version* et de l'indice dans l'environnement. Le nommage du lien par le seul indice dans l'environnement se révéle trop primitif. Prenons l'exemple d'une entrée *i* de l'environnement qui contient un lien pointant vers le module *M*. A cet instant, l'entrée *i* désigne le module M. Ce lien est détruit de l'environnement. L'entrée *i* est donc invalide. Le module effectue ensuite une duplication de lien et comble du hasard un lien vers le module *N* est copié à l'entrée *i*. L'entrée *i* désigne maintenant le module *M*. Si le module stocke ses noms de modules dans des variables internes, cela permet alors de désigner avec un même nom,

deux modules différents.
Tout nouveau lien placé sur l'entrée entraine l'incrémentation du champ *Version*. Initialement, ce champ est initialisé a 0.

- *LeLien* désigne le lien qui se trouve dans l'environnement.

Remarque

La limitation sur la taille de l'environnement pose des problèmes. Des modules catalogues (répertoire, nom symbolique) auront un nombre limité d'entrées. Des implantations avec environnement variable sont imaginables (A partir de listes chaînées). D'où une gestion mémoire plus complexe.

3.3.2.3 L'identificateur de liens

Le module manipule son environnement et les liens par des identificateurs. Pour le module, l'identificateur est vu comme un entier sur quatre octets. Au niveau de l'agence Liens, cet identificateur est représenté par le type *ModuleId*:

```
typedef union
            {
             unsigned int IdExterne ;
             struct
                   {
                    unsigned short int Version ;
                    unsigned short int Index ;
                   } IdInterne ;
            } ModuleId ;
```

- *IdExterne* est l'identificateur de lien manipulé par le module.
  Les primitives de l'agence Liens utilisent la représentation interne *IdInterne* composée de.
  - Le champ *Index* désigne le numéro de l'entrée du lien dans l'environnement.
  - Le champ *Version* a été obtenu par le champ *Version* de l' environnement.

Les primitives de l'agence Liens ont souvent en paramétres un identificateur de lien. Toute primitive se doit de vérifier la validité de cet identificateur. Un identificateur de lien est valide si *Index* est inférieur à *SizeOfEnvironnement*. De plus les champs *Version* de l'identificateur de lien et de l'environnement doivent correspondre. L'entrée de l'environnement ne doit pas être *Vide*.
La fonction *GoodLink* (ligne xxx) teste si un *IdLien* est valide:

```
char GoodLink (IdLien,Cont,Index)
 unsigned int IdLien ;
 ContexteTC *Cont ;
 ShortInt *Index ;
{
 ModuleId M ;

 M.IdExterne = IdLien ;
 if (M.IdInterne.Index > SizeOfEnvironnement)
    return FALSE ;
    else {
          *Index = M.IdInterne.Index ;
          if ((*(Cont->Environnement)) [*Index].EtatEntree == Vide)
             return FALSE ;
             else
                return (M.IdInterne.Version == (*(Cont->Environnement)) [*Index].Ver-
sion) ;
```

```
        }
}
```

- *IdLien* est l'identificateur de lien à contrôler.
- Une tâche SPART possède un contexte qui mémorise les données caracteristiques de la tâche. *Cont* est le contexte du module contenant l'environnement.
- *Index* est retournée par la fonction *GoodLink*. C'est l'indice où est rangé le lien dans l'environnement.

3.3.2.4 Lien dans le message

La structure *LienInMessage* définit la structure de donnée nécessaire pour mémoriser les liens se trouvant dans un message OMPHALE. Le type *LienInMessage* définit un élément de la liste des liens se trouvant dans le message. On apporte donc ici aucune limite au nombre de liens accompagnant le message. Les liens dans un message sont donc chaînés entre eux. Le lien de tête est repéré dans la structure *MessUtilisateur* décrite dans le paragraphe suivant.

```
typedef struct LIENINMESSAGE
              {
               short int Index ;
               Lien l ;
               struct LIENINMESSAGE *Suivant ;
              } LienInMessage ;
```

- Le champ *l* représente le lien que l'utilisateur intégre au message.
- *Index* désigne l'index dans l'environnement où se trouvait le lien. Ce champ est utile dans deux situations:
  1. A la destruction d'un message (primitives *DeleteMessage* et *DeleteAllMessages*), les liens du messages sont restitués à l'environnement. Ce champ permet ainsi de restituer le lien à son environnement.
  2. A la reprise d'un lien du message (primitive *TakeAgainLink*), le module retire un lien d'un message de sa zone d'émission. L'entrée de l'environnement du lien est repositionnée.
- *Suivant*: Pointeur sur le lien suivant dans le message.

3.3.3 Les structures gérées par l'agence Messages

3.3.3.1 Le message OMPHALE

Les modules OMPHALE émettent des messages. Au niveau du noyau, le message OMPHALE est défini avec la structure *MessUtilisateur*. Cette structure se retrouve à la fois dans la zone d'émission ZE et dans la zone de réception ZR.

```
typedef struct
                {
                 char TypeMessage ;

                 unsigned int Site ;
                 xx_id Module ;

                 short int IdQuai ;

                 short int Longueur ;
                 char *Utile ;

                 LienInMessage *FirstLien ;
                } MessUtilisateur,*PtMessUtilisateur ;
```

Cette structure est composée de deux types de champs.

- Les champs utilisés par le système de transport. Ils permettent l'acheminement complet du message vers le module destinataire.
  - Le champ *TypeMessage* sera étudié ultérieurement.
  - Les champs *Site* et *Module* désignent le module destinataire. Ces champs sont initialisés à partir du nom unique du lien ayant servi à l'émission.
  - Le champ *IdQuai* désigne le numéro du quai où le message sera déposé. (numéro de la file d'attente dans la zone de réception du module destinataire).
- Le deuxième type de champ a trait au contenu du message proprement dit.
  - Le champ *Utile* désigne le contenu du message . C'est une zone de longueur variable. Le champ *Longueur* indique la longueur de la zone pointée par *Utile*. Cette zone contient des valeurs transmises en paramétre au module destinataire.
  - A chaque message, on associe la liste des liens se trouvant dans celui-ci. Le champ *FirstLien* pointe sur le premier lien de cette liste. Les primitives de l'agence Liens TransferLink, et TakeAgainLink ajoutent ou suppriment des liens de cette liste.

3.3.3.2 La zone d'émission

La zone d'émission (ZE) contient les messages OMPHALE crées par le module (primitive *CreateMessage*) au cours de la dernière phase d'activité. Le nombre de messages de la zone d'émission est à priori indéfini d'où l'utilisation d'une liste chaînée. Les messages de la zone d'émission possédent un numéro *Name*. La zone d'émission est triée en ordre croissant de numéro de message. La création d'un nouveau message se fait à la fin de la zone d'émission d'où le pointeur de queue. Nous ne réutilisons pas les numéros de messages détruits.
Cette zone d'émission est localisée dans le segment système global et ce pour la rendre accessible du système de transport. Le type *MessBuffEmmission* définit un élément de la liste chaînée des messages OMPHALE.

```
typedef struct
          MESSBUFFEMISSION
                        {
                         short int Name ;
                         short int Index ;
                         PtMessUtilisateur PtMUtil ;
                         struct MESSBUFFEMISSION *Suivant ;
                        }
          MessBuffEmission, *PtMessBuffEmission ;
```

- *Name* : C'est le nom local du message OMPHALE. C'est l'identificateur qui a été retourné par la primitive *CreateMessage*. Le module doit utiliser ce nom pour placer des liens dans ce message (*TransferLink*) ou pour la destruction du message de la zone d'émission.
- *PtMUtil*: pointeur sur la structure *MessUtilisateur* qui a été décrite précédemment.
- *Suivant*: Pointeur sur le message suivant de la zone d'émission.
- *Index*: Ce champ désigne l'identificateur du lien ayant servi à la création du message. A la destruction du message, le noyau doit être capable de retrouver le lien ayant servi à l'émission du message. Si un service est à utilisation unique, l'émission d'un message rend ce service inutilisable. Si ce message est ensuite détruit, le service redevient utilisable.

La zone d'émission est constituée par deux pointeurs qui sont déclarés dans la structure de *MessageAttendre*. Cette structure est partagée entre TN et TC. Elle comprend deux pointeurs sur le premier et le dernier message de la zone d'émission.

3.3.4 Les structures gérées par l'agence routage

3.3.4.1 La zone de réception

La zone de réception contient les messages qui n'ont pas encore été traités par le module. Seule la tâche TN accéde à cette zone.

```
typedef struct MESSAGESURQUAI
             {
              unsigned int DateArrivee ;
              PtMessUtilisateur PtMUtil ;
              struct MESSAGESURQUAI *Suivant ;
             } MessageSurQuai ;

typedef struct
             {
              MessageSurQuai *Tete,*Queue ;
             } TYP_QUAI ;

TYP_QUAI Quai [MAXSERVICES] ;
```

- Un message sur un quai se caractérise par les informations suivantes:
  - Les messages sont datés à l'arrivée dans la zone de réception. Le champ *DateArrivee* contient cette date. Cette date sera utilisée par le noyau pour le choix du service à activer.
  - Le champ *PtMUtil* désigne le message OMPHALE.
  - Le champ *Suivant* est un pointeur sur le message OMPHALE suivant situé sur le même quai.
- Un quai est organisée en file FIFO. Un quai *TYP_QUAI* se définit donc comme deux pointeurs:
  - Un pointeur sur le premier message du quai:*Tete*
  - Un pointeur sur le dernier message du quai:*Queue*

  Les messages arrivant sur un quai sont ajoutés en queue. La suppression d'un message se fait en tête.
- La zone de réception est structurée comme un tableau de *Quais*. La taille du tableau est fixée par MAXSERVICES. Cette constante est initialisée à 16.

3.3.4.2 La communication TC-TN

La communication entre TC et TN se fait par:

- Deux boîtes aux lettres MBN et MBC.
- Des messages SPART qui font au plus 8 octets. La mémoire commune est donc nécessaire pour la communication d'information de taille supérieure.

A la naissance du module, TN alloue un emplacement mémoire du type *MessageAttendre* qui servira exclusivement à cette communication. Cette zone sera libérée par TN à la mort du module (passage à l'état **Mourant**). Cette zone est ensuite utilisée pour toutes les inter-actions entre TC et TN et en particulier au niveau du *Wait*.

```
typedef struct
              {
               char Normal ;
               unsigned short int QuaisAOuvrir ;
               char *MessageRecu ;
               short int Longueur ;
               short int NombreLiensRecus ;
               int LesLiensRecus [SizeOfEnvironnement] ;
               MessBuffEmission *Debut,*Fin ;
              } MessageAttendre,*PtMessageAttendre ;
```

- *Normal* est un champ booléen. Si *Normal* est à TRUE, l'attente correpond à une attente normale. Dans le cas contraire, elle provient d'un trap provoqué par une erreur de programmation.
- *QuaisAOuvrir* est positionné par TC (*SetServ*, *ResServ* et *PutServ*). Elle indique à TN les quais que le module désire ouvrir. TN pourra choisir un message sur l'un des quais spécifiés dans *QuaisAOuvrir* pour la réactivation de TC.
- *MessageRecu* désigne le message OMPHALE reçu pour la phase d'activité courante. *Longueur* donne sa longueur en octets.
- *NombreLiensRecus* indique le nombre de liens reçus avec le message OMPHALE.
- *LesLiensRecus* est un tableau qui contient les identificateurs de liens qui ont été reçus avec le message OMPHALE.
- *Debut* et *Fin* sont des pointeurs vers le premier et le dernier message de la zone d'émission.

3.3.5 Le contexte du module

Les informations propres à chaque module sont mémorisées dans les contextes des tâches TC et TN. Pour chaque tâche, SPART associe un contexte. Ce contexte contient les informations propres à la tâche. Ce contexte peut être étendu en fonction de l'application.

3.3.5.1 Le contexte de TN

```
typedef struct {
                xx_id MbxIdTN,MbxIdTC,SegIdSyst ;
                unsigned int BaseSyst ;
                int DuplicateCount ;
                struct MESSAGEATTENDRE *MessAtt ;
                char Etat      ;
                Env *Environnement ;
                unsigned short int QuaisOuverts,QuaisOuIlyaDesMessages ;
                TYP_QUAI Quai [MAXSERVICES] ;
               } *ContexteOmphale,StructContexteOmphale ;

typedef struct {
                ctxt ctx_sys ;
                ContexteOmphale CtxSysOmphale ;
               } ContexteTN ;
```

Le contexte de la tâche TN se compose de deux champs:

- *ctx_sys*: Informations du contexte standard des tâches SPART. Ce champ est décrit dans l'annexe B.
- *CtxSysOmphale*

Le type *ContexteOmphale* définit toutes les informations nécessaires au noyau OMPHALE au niveau de la tâche TN.

- *MbxIdTC* et *MbxIdTN* sont les identificateurs des boîtes aux lettres MBN et MBC.
- *SegIdSyst* et *BaseSegSyst* sont respectivement l'identificateur du segment système et son adresse de début.
- *DuplicateCount* désigne le compteur de références du module dans le système.
- *MessAtt* est un pointeur sur la zone de communication entre TC et TN.
- *Etat* précise l'état actuel du module. (**Actif, Attente, Mourant**).
- *Environnement* est un pointeur sur l'environnement du module.

- Le champ *Quai* désigne la zone de réception du module.
- Le champ *QuaisOuIlyaDesMessages* indique les quais qui contiennent des messages OMPHALE.

3.3.5.2 Le compteur de duplication

Pour chaque module, nous désignons par compteur de duplication le nombre de référence du module existant dans le système. C'est-à-dire le nombre de liens référençant le module. Ce compteur est implémenté comme une variable entière qui se trouve dans le contexte du module. (Voir le contexte de la tâche TN). Son incrémentation fait suite à la réception d'un message du type *Duplication* sur la boîte aux lettres MBN. Sa décrémentation est la conséquence de la réception d'un message de type *Destruction.*

3.3.5.3 Le contexte de TC

Le type *ContexteTC* définit le contexte de la tâche TC. Le champ *ctx_sys* est utilisé par le noyau SPART. Les autres champs sont gérés par le noyau OMPHALE.

```
typedef struct {
                ctxt ctx_sys ;

                xx_id MbxIdTN,MbxIdTC,SegIdSyst,SegIdGT,MbIdGT ;
                unsigned int BaseSyst,BaseGT ;
                struct MESSAGEATTENDRE *MessAtt ;

                short int NextNameMessage ;
                Env *Environnement ;
               } ContexteTC ;
```

- *MbxIdTC* et *MbxIdTN* sont les identificateurs SPART des boîtes aux lettres MBC et MBN.
- *SegIdSyst* est l'identificateur SPART du segment système. *BaseSyst* est l'adresse de début du segment système.
- Les tâches GT sont responsables de la création de module sur le site. La communication avec ces tâches se fait par la boîte aux lettres *MbIdGT* et un segment mémoire *SegIdGT.*
- Le champ *MessAtt* désigne un pointeur sur la zone de communication entre TC et TN.
- *NextNameMessage* est le nom du prochain message OMPHALE crée par le module. (primitive *CreateMessage*).
- *Environnement* est un pointeur sur l'environnement du module.

3.4 Les primitives du noyau

La spécification des primitives du noyau se trouve dans l'annexe C. Pour chaque primitive, cette annexe décrit son action sur le système, ses paramètres (en C et MODULA-2) ainsi que les erreurs qu'elles retournent.
L' implantation des primitives des agences Liens, Messages et Routage est décrite dans les annexes E, F et G.
Les primitives de l'agence Liens manipulent l'environnement du module. Elles sont exécutées par la tâche TC du module OMPHALE.
Les primitives de l'agence Messages agissent sur la zone d'émission du module. Elles sont appelées par la tâche TC pendant la phase d'activité.
Les primitives de l'agence Routage sont exécutées par la tâche TN. La tâche TN manipule la zone de réception au moyen de ces primitives.

AGENCE LIENS

Primitive	Description	Annexe
DuplicateLink	Dupliquer un lien	E.1
DestroyLink	Détruire un lien	E.2
RetrieveLink	Récupérer un lien détruit	E.3
TransferLink	Transférer un lien dans un message	E.4
TakeAgainLink	Retirer un lien d'un message	E.5
SetNoUse	Supprimer des services accessibles par un lien	E.6
SetOneUse	Rendre un service utilisable une fois	E.7
SetNoDups	Rendre un lien non duplicable	E.8
StateLink	Etat du lien (services accessibles)	E.9
NumberLinkReceived	Nombre de liens du message	E.10
IdLinkReceived	Identificateur du lien du message	E.11

AGENCE MESSAGES

Primitive	Description	Annexe
CreateMessage	Création d'un message dans la ZE	F.1
DeleteMessage	Destruction d'un message de la ZE	F.2
DeleteAllMessages	Destruction des messages de la ZE	F.3
SetServ	Validation de services	F.4
PutServ	Ajout de services validés	F.5
ResServ	Fermeture services	F.6
ReadServ	Lecture des services validés	F.7
SizeMessage	Taille du message reçu	F.8
LastMessage	Lecture du message reçu	F.9
Wait	Attente Message (fin de la phase d'activité	F.10
Failure	Erreur de programme (handler)	F.11
InitBuffer	Initialisation de la ZE	F.12

AGENCE ROUTAGE

Primitive	Description	Annexe
Dupliquer	Duplication de lien	G.1
Detruire	Destruction de lien	G.2
Externe	Message OMPHALE	G.3
Attendre	Attente du module (TC bloquée)	G.4

3.5 Réalisation du schéma d'exécution

Dans cette section, nous décrivons les primitives du noyau qui ont trait au schéma d'exécution. Le module termine sa fin d'activité par la primitive *Wait* de l'agence Messages.

3.5.1 Wait

La primitive Wait marque la fin de la phase d'activité du module OMPHALE. La tâche TC exécute donc la primitive *Wait* de l'agence Message.

```
Status Wait ()
{
 status Service ;
 ContexteTC *ContexteModuleCourant ;
 char MessSpart [MESS_LENGTH] ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;

 MessSpart [0] = CodeAttendre ;
 ContexteModuleCourant->MessAtt->Normal = TRUE ;
 send (MessSpart,ContexteModuleCourant->MbxIdTN,0) ;

 receive  (MessSpart,ContexteModuleCourant->MbxIdTC,INFINITE) ;

 if (MessSpart [0] == CodeDetruire)
     FinTacheTC (ContexteModuleCourant) ;

 ContexteModuleCourant->NextNameMessage = 0 ;
 memcpy (&Service,MessSpart+1,sizeof(status)) ;
 return Service ;
}
```

Pendant la phase d'activité, le module a déposé (primitive *CreateMessage*) dans sa ZE des messages OMPHALE en vue d'un transfert. L'appel à la primitive Wait marque la fin de la phase d'activité du module.

- TC émet un message SPART vers TN lui précisant que le module se met en attente (TC est en attente). TC attend une réponse sur la boîte aux lettres MbxIdTC.
- TN récupére le message émis par TC. Les actions exécutées par TN seront étudiées plus précisement dans la section routage du présent chapitre.
  - TN est chargée de l'émission des messages de la ZE. Les messages distants sont passés au système de transport qui assure le transfert des messages. Les messages locaux sont directement envoyés au module (la tâche TN) destinataire.
  - Dans certaines situations, TN décide de la fin du module. Si tous les quais sont fermés, le module n'est plus reactivable. De plus si le module n'est plus référencé dans le système et que les quais ouverts sont vides, le module ne sera jamais réactivé. Un message de destruction est alors envoyé à la tâche TC.
  - TN choisit un service et un message sur un des quais ouverts. Un message SPART de réactivation est alors émis à destination de la tâche TC.
- TC reçoit le message SPART de TN. Deux possibilités:
  - Un message de Destruction. C'est la fin du module et la tâche TC doit donc s'arrêter. La fonction FinTacheTC stoppe l'exécution de TC et assure la destruction de la boîte aux lettres *MbxIdTC*.

```
void FinTacheTC (Cont)
  ContexteTC *Cont ;
{
 xx_id MbIdRelai ;
 MessGT *PtName ;
 char MessSpartRelai [MESS_LENGTH] ;
 general_address ga ;
 task_obj *task = NIL ;

 find (GLOBAL,"BLREL",&MbIdRelai) ;
 alloc_memory (Cont->SegIdGT,sizeof(MessGT),&PtName) ;
 PtName = (MessGT *) (Cont->BaseGT + (unsigned int) PtName) ;
 PtName->Type = Delete ;
 task = (task_obj *) access_obj (Cont->ctx_sys.task_id) ;
 PtName->mbn = task->prog_id ;
 gen_addr (PtName,&ga) ;
 memcpy (MessSpartRelai,&ga,sizeof(general_address)) ;
 send (MessSpartRelai,MbIdRelai,0) ;

 delete_mbx (Cont->MbxIdTC) ;

 exit_task () ;
}
```

    - La tâche *Relai* est responsable de la gestion des tâches calcul TC. Elle assure la création de l'ensemble des tâches calculs du site. Son fonctionnement sera plus précisement décrit dans la section Création et destruction de module. La tâche *Relai* est prévenue par la tâche TC. L'image binaire de la tâche pourra éventuellement être déchargée.

- La boîte aux lettres *MbxIdTC* est détruite.
- La tâche TC s'arrête (*exit_task*).

– Un message de continuation. Le module est actif et le message reçu contient le numéro du service choisi par TN. La zone de communication MessAtt contient le message reçu ainsi que les liens accompagnant le message.

3.5.2 Attendre

Lorsque le module s'est mis en attente, TC a émis un message SPART à l'intention de la tâche TN. A la réception de ce message, TN exécute la primitive *Attendre* de l'agence Routage.

```
status Attendre ()
 {
  status cr ;
  unsigned int Site, Pemis, i ;
  LienInMessage *Ptl ;
  char MessSpart [MESS_LENGTH] ;
  PtMessBuffEmission Ptm, P ;
  ContexteTN *Cont ;
  ContexteOmphale PCO ;

  Cont = (ContexteTN *) tache_courante () ;
  PCO = Cont->CtxSysOmphale ;

  if (PCO->MessAtt->Longueur)
    {
     PCO->MessAtt->MessageRecu = (char *)
                                 AddAbsolTN (PCO->MessAtt->MessageRecu) ;
     DisposeGlobalTN (PCO->MessAtt->MessageRecu,PCO->MessAtt->Longueur) ;
    } ;

  PCO->MessAtt->Longueur = 0 ;
  PCO->MessAtt->MessageRecu = NIL ;

  if (PCO->MessAtt->Normal)
    {
     PCO->QuaisOuverts = PCO->MessAtt->QuaisAOuvrir ;
     Ptm = (PtMessBuffEmission) AddAbsolTN (PCO->MessAtt->Debut) ;
     PCO->MessAtt->Debut = NIL ;

     MessSpart [0] = CodeExterne ;
     while (Ptm)
          {

           Ptm->PtMUtil = (PtMessUtilisateur) AddAbsolTN (Ptm->PtMUtil) ;
           Ptl = (LienInMessage *) AddAbsolTN (Ptm->PtMUtil->FirstLien) ;

           while (Ptl)
                {
                 (*(PCO->Environnement)) [Ptl->Index].EtatEntree = Vide ;
                 Ptl = (LienInMessage *) AddAbsolTN (Ptl->Suivant) ;
                } ;
           Pemis = AddRelatTN (Ptm->PtMUtil) ;
           memcpy (MessSpart+1,&Pemis,sizeof(unsigned int)) ;
           Site = Ptm->PtMUtil->Site ;

           if (Site == MyHouse)
              send (MessSpart,Ptm->PtMUtil->Module,0) ;
```

```
            else EnvoiST (Ptm->PtMUtil) ;

          P = Ptm ;
          Ptm = (PtMessBuffEmission) AddAbsolTN (Ptm->Suivant) ;
          DisposeGlobalTN (P,sizeof(MessBuffEmission)) ;
         }
      }
  else {
        PCO->QuaisOuverts = 0 ;
        DetruireMessageZE (PCO) ;
       }

 for (i=0;i<SizeOfEnvironnement;i++)
    if ( ((*(PCO->Environnement)) [i].EtatEntree == LienADetruire))
      {
       (*(PCO->Environnement)) [i].EtatEntree = Vide ;
       DestroyNet ((*(PCO->Environnement)) [i].LeLien.SiteName,
                   (*(PCO->Environnement)) [i].LeLien.Destinataire) ;
      }

 if (PCO->QuaisOuverts & PCO->QuaisOuIlyaDesMessages)
   {
    ActiverTC (PCO) ;
    return GOOD ;
   }
  else if (PCO->QuaisOuverts)
         {if (PCO->DuplicateCount == NombreAutoReference (PCO))
            {
             DevientMourant (PCO) ;
             return GOOD ;
             }
           PCO->Etat = EnAttente ;
           return GOOD ;
         }
       else {
             DevientMourant (PCO) ;
             if (PCO->DuplicateCount == 0)
                DetruireTN (PCO) ;
             return GOOD ;
            }
}
```

- Au terme de cette phase d'activité, La zone mémoire allouée pour le message est libérée.
- La zone d'émission est parcourue. Pour chaque message de cette zone d'émission, le traitement suivant est exécuté:
  - Les liens sont réellement retirés de l'environnement. Le champ *EtatEntree* est positionné à *Vide*.
  - Le message est émis vers la station de transport si le destinataire n'est pas local (fonction *EnvoiST*). Le noyau NERF conserve l'identificateur de la boîte aux lettres de la station de transport. Un message SPART est donc émis vers la station de transport.

Si le destinataire est un module local, un message SPART est émis vers la tâche TN du destinataire.

- TN détruit l'ensemble des liens de l'environnement dont la destruction avait été demandée (primitive *DestroyLink*). La fonction *DestroyNet* assure la décrémentation du compteur de duplication du module par un envoi de message.
- TN doit ensuite analyser le nouvel état du module.
  - S'il existe un quai Ouvert qui ne soit pas vide, le module (la tâche TC) peut être réactivé. La fonction *ActiverTC* réveille la tâche TC.
  - S'il n'existe plus de référence vers le module ou tous les quais du modules sont fermés. Le module de OMPHALE ne peut être réactivé. Le module devient alors **Mourant.**
  - Le passage à l'état **Mourant** déclenche l'exécution de la fonction *DevientMourant*

```
void DetruireMessagesQuai (P)
 MessageSurQuai *P ;
{
 MessageSurQuai *pc,*pp ;

 for (pc = P ; (pc - NIL) ;)
   {
    DetruireMessageRecu (AddAbsolTN (pc->PtMUtil)) ;

    pp = pc ;
    pc = pc->Suivant ;
    DisposeLocalTN (pc,sizeof(MessageSurQuai)) ;
   }
}

void DetruireZR (PCO)
 ContexteOmphale PCO ;
{
 int i = 0 ;

 for (i = 0 ; i < MAXSERVICES; i++)
   DetruireMessagesQuai (PCO->Quai [i]) ;
}

void DetruireTC (PCO)
 ContexteOmphale PCO ;
{
 status cr ;
 char MessFin [MESS_LENGTH] ;

 MessFin [0] = CodeDetruire ;
 send (MessFin,PCO->MbxIdTC,0) ;
}


void DevientMourant (PCO)
ContexteOmphale PCO ;
```

```
{
 int i ;

 PCO->Etat = Mourant ;
 DetruireTC (PCO) ;

 for (i=0;i< SizeOfEnvironnement;i++)
   if ((*(PCO->Environnement)) [i].Occupe)
     DestroyNet ((*(PCO->Environnement)) [i].LeLien.SiteName,
                 (*(PCO->Environnement)) [i].LeLien.Destinataire
                ) ;
 DisposeGlobalTN (PCO->Environnement,sizeof (Env)) ;
 DisposeGlobalTN (PCO->MessAtt,sizeof(MessageAttendre)) ;
 DetruireZR (PCO) ;
}
```

- *DevientMourant* débute par la destruction de la tâche TC. La procédure *DetruireTC* effectue cette destruction en envoyant un message stop à la tâche TC.
- Les liens se trouvant encore dans l'environnement sont détruits.
- L'espace mémoire utilisé pour l'environnement et la zone de communication TC-TN est restitué.
- La fonction *DetruireZR* détruit les messages restant dans la zone de réception. (Ainsi que les liens accompagnant ces messages).

– Lorsque le compteur de duplication est nul et que le module est mourant, le module disparait. La procédure *DetruireTN* est alors appelée.

– Lorsque le champ *Normal* est positionné à FALSE, les messages de la zone d'emission sont détruits par la fonction *DetruireMessageZE*. Le champ *QuaisOuverts* est positionné à 0. Le module va donc ainsi passer à l'état **Mourant**.

```
void DetruireMessageZE (PCO)
ContexteOmphale PCO ;
{
 MessBuffEmission *pc,*pp ;
 PtMessUtilisateur Ptu ;

 for (pc = (MessBuffEmission *) AddAbsolTN (PCO->MessAtt->Debut) ;
      (pc - NIL) ;)
   {
    Ptu = (PtMessUtilisateur) AddAbsolTN (pc->PtMUtil) ;
    DetruireMessageRecu (Ptu) ;

    pp = pc ;
    pc = (MessBuffEmission *) AddAbsolTN (pc->Suivant) ;
    DisposeGlobalTN (pp,sizeof (MessBuffEmission)) ;
   } ;
}
```

– Le traitement est est analogue à celui de la primitive *DeleteAllMessages* de l'agence Messages.

3.5.3 Externe

Le module a reçu un message OMPHALE provenant d'un autre module. Cela se traduit par la réception au niveau de la tâche TN d'un message SPART contenant l'adresse du MESSAGE OMPHALE. La tâche TN exécute alors la primitive *Externe* de l'agence Routage.

```
status Externe (MessSpart)
char MessSpart [] ;
{
 MessageSurQuai *Pmsq ;
 PtMessUtilisateur Ptu ;
 ContexteTN *Cont ;
 ContexteOmphale PCO ;

 Cont = (ContexteTN *) tache_courante () ;
 PCO = Cont->CtxSysOmphale ;

 memcpy (&Ptu,MessSpart+1,sizeof (unsigned int)) ;
 Ptu = (PtMessUtilisateur) AddAbsolTN (Ptu) ;
 if (PCO->Etat == Mourant)
   {
    DetruireMessageRecu (Ptu) ;
    return GOOD ;
   }
  else
     {
      Pmsq = (MessageSurQuai *) NewLocalTN (sizeof(MessageSurQuai)) ;
      Pmsq->PtMUtil = Ptu ;
      Pmsq->Suivant = NIL ;
      get_time (&Pmsq->DateArrivee) ;
      if (PCO->Quai [Ptu->IdQuai].Tete == NIL)
        {
         PCO->Quai [Ptu->IdQuai].Tete = PCO->Quai [Ptu->IdQuai].Queue = Pmsq ;
         PCO->QuaisOuIlyaDesMessages |= SetOneBitTN (Ptu->IdQuai) ;
        }
       else {
             PCO->Quai [Ptu->IdQuai].Queue->Suivant = Pmsq ;
             PCO->Quai [Ptu->IdQuai].Queue = Pmsq ;
            } ;
      if ((PCO->Etat == EnAttente)
          &&
            (
             PCO->QuaisOuverts
               &
             PCO->QuaisOuIlyaDesMessages
            )
         )
        ActiverTC (PCO) ;
        return GOOD ;
      }
}
```

- Si le module est **mourant**, le message est détruit ainsi que les liens l'accompagnant. (fonction *DetruireMessageRecu*).

```
void DetruireMessageRecu (P)
PtMessUtilisateur P ;
{
 if (P->Longueur)
    DisposeGlobalTN (AddAbsolTN (P->Utile),P->Longueur) ;
 DetruireLienMessage (P->FirstLien) ;
 DisposeGlobalTN (P,sizeof(MessUtilisateur)) ;
}
```

Cette fonction libére l'espace mémoire du message. Les liens contenus dans le message sont détruits par la fonction *DetruireLienMessage*.

```
void DetruireLienMessage (P)
LienInMessage *P ;
{
 LienInMessage *pc,*pp ;

 for (pc = (LienInMessage *) AddAbsolTN (P);(pc - NIL);)
   {
    DestroyNet (pc->l.SiteName,pc->l.Destinataire) ;

    pp = pc ;
    pc = (LienInMessage *) AddAbsolTN (pc->Suivant) ;
    DisposeGlobalTN (pp,sizeof(LienInMessage)) ;
   }
}
```

La liste des liens du message est parcourue complétement. Un message de destruction de référence est émis à l'intention de la tâche TN du module pointé par le lien (fonction *DestroyNet*). L'espace mémoire utilisé pour le stockage du lien dans le message est libéré.

- Si le module n'est pas **mourant** (**Actif** ou **Attente**), le message est placé sur le quai. Le message est daté (fonction *get_time*). Le message est chaîné aux autres messages du quai. Si le module est en atente et que l'un des quais ouverts n'est pas vide, le module est reactivé. (fonction *ActiverTC*)

```
void ActiverTC (PCO)
ContexteOmphale PCO ;
{
 status cr,QuaiChoisi ;
 char MessSpart [MESS_LENGTH] ;
 MessageSurQuai *Pmsq ;
 LienInMessage *Ptl ;

 QuaiChoisi = QuaiAvecLeVieuxMessage (PCO) ;

 PCO->MessAtt->MessageRecu = PCO->Quai [QuaiChoisi].Tete->PtMUtil->Utile ;
 PCO->MessAtt->Longueur = PCO->Quai [QuaiChoisi].Tete->PtMUtil->Longueur ;
```

```
PCO->MessAtt->NombreLiensRecus = 0 ;
Ptl = (LienInMessage *)
          AddAbsolTN PCO->Quai [QuaiChoisi].Tete->PtMUtil->FirstLien) ;
while (Ptl)
   {
    PCO->MessAtt->NombreLiensRecus ++ ;
    PCO->MessAtt->LesLiensRecus [PCO->MessAtt->NombreLiensRecus] =
                      Ajouter (PCO,Ptl->l) ;
    Ptl = (LienInMessage *) AddAbsolTN (Ptl->Suivant) ;
   } ;

DisposeGlobalTN (PCO->Quai [QuaiChoisi].Tete->PtMUtil,
                 sizeof(MessUtilisateur)
                ) ;

Pmsq = PCO->Quai [QuaiChoisi].Tete ;
PCO->Quai [QuaiChoisi].Tete = PCO->Quai [QuaiChoisi].Tete->Suivant ;
DisposeLocalTN (Pmsq,sizeof (MessageSurQuai)) ;

if (PCO->Quai [QuaiChoisi].Tete == NIL)
   PCO->QuaisOuIlyaDesMessages &= (~SetOneBitTN (QuaiChoisi)) ;

MessSpart [0] = CodeExecution ;
memcpy (MessSpart+1,&QuaiChoisi,sizeof(status)) ;

cr = send (MessSpart,PCO->MbxIdTC,0) ;
PCO->Etat = EnCours ;
}
```

Les liens du message sont ajoutés (fonction *Ajouter* à l'environnement du module. Les identificateurs de liens sont placés dans le tableau *LesLiensRecus*.
Parmi les files d'attentes ouvertes et qui ne sont pas vides, la fonction *QuaiAvecLeVieuxMessage* détermine le message le plus ancien de la zone de réception. Elle balaye la zone de réception en comparant les dates d'arrivée des différents messages. Elle choisit le quai avec le message le plus ancien (la plus petite date).

```
status QuaiAvecLeVieuxMessage (PCO)
ContexteOmphale PCO ;
{
 status i = 0,QuaiGagnant=0,j ;
 unsigned int HeureDuGagnant = (-1) ;


 for (;i < MAXSERVICES;i++)
    {
     if (PCO->Quai[i].Tete - NIL)
        {
         if ((PCO->Quai[i].Tete->DateArrivee < HeureDuGagnant)
              &&
             (PCO->QuaisOuverts  SetOneBitTN (i)
            )
             {
```

```
                QuaiGagnant = i ;
                HeureDuGagnant = PCO->Quai [i].Tete->DateArrivee ;
               }
          }
     }
  return QuaiGagnant ;
}
```

Les liens accompagnant le message sont ajoutés dans l'environnement. Les identificateurs de liens sont placés dans les tableau *LesLiensRecus*.

L'espace mémoire utilisé par le message dans la zone de réception est libéré.

La tâche TC est réactivée par un message SPART envoyé sur *MbxIdTC*. Le message contient le numéro du service (quai). La structure *MessAtt* contient les références du messages OMPHALE ainsi que les identificateurs des liens se trouvant dans le message.

3.6 Le code de la tâche TN

```
main ()
 {
  xx_id mbn ;
  char MessSpart [MESS_LENGTH] ;

  receive (MessSpart,0,0) ;
  mbn = InitTN (MessSpart) ;

  while (TRUE)
      {
       receive (MessSpart,mbn,INFINITE) ;
       switch (MessSpart [0])
            {
             case CodeDuplication : Dupliquer () ;
                                    break ;
             case CodeDetruire    : Detruire () ;
                                    break ;
             case CodeAttendre    : Attendre () ;
                                    break ;
             case CodeExterne     : Externe  (MessSpart) ;
                                    break ;
             default : printf ("TN erreur Message %d\n",MessSpart [0]) ;
            }
      }
 }
```

1. La tâche TN exécute d'abord la primitive *initTN* qui initialise le module et son contexte. La primitive *InitTN* renvoie l'identificateur de la boîte aux lettres MBN.

2. Le traitement suivant consiste à lire un message sur la boîte aux lettres *mbn*. Le type du message se trouve dans le premier octet du message. Ce type permet d'appeler la primitive de l'agence routage correspondante.

3.7 Création et destruction de module OMPHALE

3.7.1 La création de module OMPHALE

La création de module OMPHALE est effectuée par les modules Gestionnaire de module du CORTEX. Le fonctionnement de ces modules sera détaillé dans le prochain chapitre. Le noyau fournit à ces modules la primitive *CreateModule*. Les opérations de création et destruction de modules font appel à plusieurs tâches:

- Les tâches *GT* assurent la création des tâches TN.
- Les tâches *Relai* assurent elles, la création des tâches TC
- La tâche *LOTN* charge le code de la tâche TN. Ce segment de code sera ensuite utilisé par les tâches *GT*.
- La tâche *MODPER* assure la création des modules Permanents, c'est à dire de modules systèmes du CORTEX qui existent au démarrage du système.

Cette primitive fait intervenir les tâches *GT* et *Relai*. Grossièrement, *GT* assure la création de la tâche TN du module OMPHALE ainsi que celle des boîtes aux lettres MBN et MBC. La tâche *Relai* crée la Tâche TC.

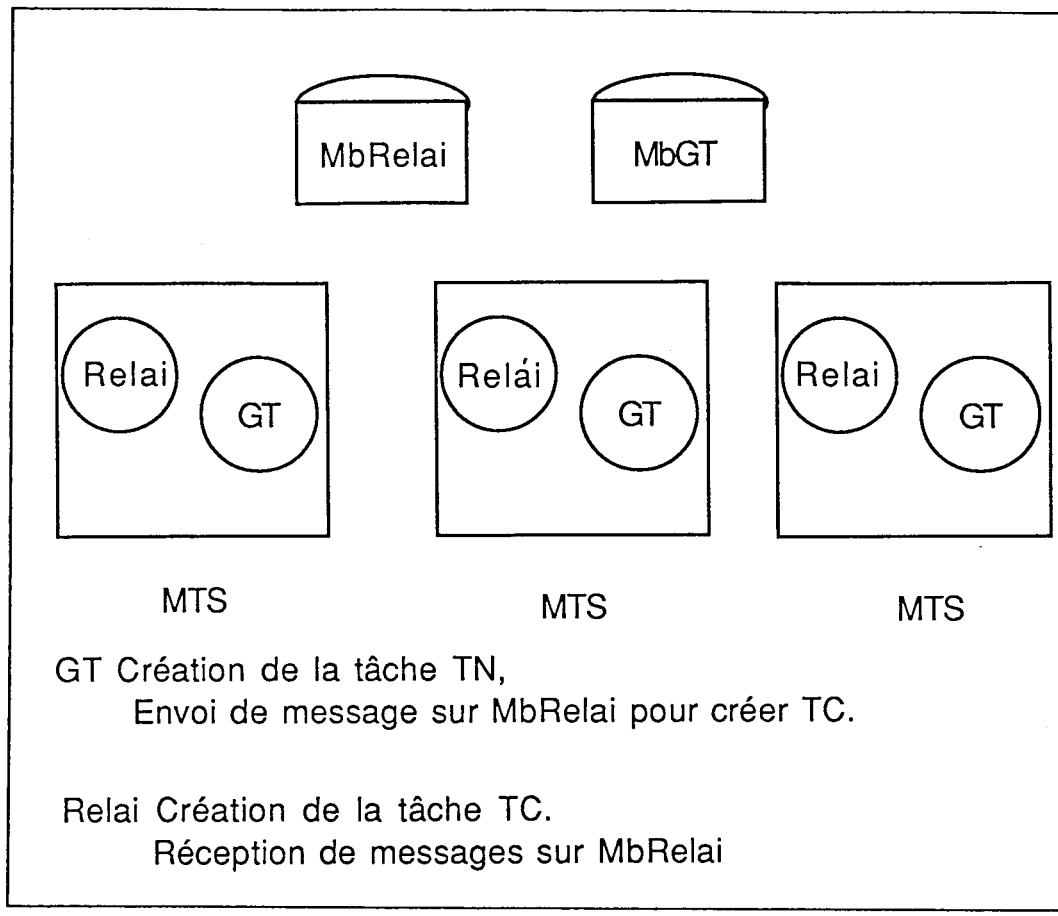


Figure 12. Les tâches GT et Relai

La configuration peut être constituée de plusieurs modules de traitement sous SPART. Le module de traitement peut alors supporter l'exécution:

- des tâches noyau (MTN). Le module de traitement exécute exclusivement des tâches noyau (TN). Le noyau chargé sur ce module doit donc inclure l'agence Routage.
- des tâches calcul (MTC). Le module de traitement supporte l'exécution de tâches calcul (TC). Le code du noyau chargé sur ce module doit donc inclure le code des agences Messages et Liens.
- Le module supporte à la fois des tâches calcul et noyau. Le noyau comprend alors le code des trois agences.

Il existe une tâche *GT* par agence Routage.
De même, il existe une tâche *Relai* par agence Message.

La création de module consiste à créer la tâche calcul sur un MTC sans savoir à priori lequel. Le même principe est appliqué pour la tâche noyau.

3.7.1.1 La primitive CreateModule

Cette primitive du noyau est exécutée par les modules gestionnaire de module. Elle a comme paramétre un nom de module et renvoie l'identificateur du lien vers le module créé. Le source de cette primitive se trouve à l'annexe F.13.

- La primitive *CreateModule* a deux paramètres qui sont:
  - *FileName*: Nom du module (fichier) contenant le code de la tâche TC.
  - *IdLien*: Identificateur du lien vers le module crée.

  La requête sera traitée par l'une des tâches *GT* du site. Un message est donc émis et les paramètres sont recopiés dans le segment *SegIdGT*.
  - Allocation mémoire dans le segment *SegIdGT*. Les paramètres de la requête y sont recopiés dans ce segment.
  - Envoi d'un message dans la boîte aux lettres *MbIdGT*. Le message sera traité par l'une des tâches *GT* du site.
- La tâche *GT* ayant traité la requête envoie sa réponse au module.
  - Le premier octet contient le status de la requête. Les deux octets suivants contiennent le numéro de la boîte aux lettres MBN du module créé. Ce numéro servira à la création du lien dans l'environnement.
  - Le lien vers le module créé est placé dans l'environnement du module. L'identificateur du lien dans l'environnement est renvoyé par le paramètre *IdLien*.

Sur cette implantation de l'architecture OMPHALE, la lecture de l'image binaire du module, l'allocation mémoire (segments de Code, Pile) sont internes au noyau. Ces fonctions sont normalement à la charge de modules système du CORTEX. Certains mécanismes du noyau SPART ne peuvent être contournés.

3.7.1.2 La tâche LOTN

Cette tâche est activée à la mise en route du site. Elle charge le code de la tâche TN en mémoire locale du M.T.Ce code sera ensuite utilisé pour la création de module par les tâches *Relai* et *MODPER*. Pour ce faire, nous utilisons le catalogue SPART sous le nom externe CODTN.

```
main ()
{
 xx_id ProgIdTN ;

 load (SYST_UNIT,"/omphale/TN",REENTRANT,&ProgIdTN) ;
 catalog (ProgIdTN,"CODTN") ;
 exit_task () ;
}
```

3.7.1.3 La tâche GT

La tâche GT (annexe H.1) assure la création de module. Elle effectue en particulier la création de la tâche TN du module. Il existe une tâche GT par module de traitement noyau (MTN). Cette création est effectuée à la demande de modules du CORTEX qu'on appelle le gestionnaire de module.

- La procédure *InitGT* initialise la tâche GT:
  - Le code de la tâche TN est chargé par la tâche LOTN. L'identificateur du code est récupéré via le catalogue SPART.
  - Il faut également récupérer les identificateurs de la boîte aux lettres des tâches *Relai* et celle des Tâches *GT*.
- Une **création** de module consiste à:
  - créer les boîtes aux lettres MBN et MBC.
  - créer et lancer la tâche TC du module. Cette fonction est assurée par une des tâches *Relai* du site.
  - Si tout s'est bien passé du côte de la tâche *Relai*, la tâche TN peut être créée et initialisée. Cette création de tâche se fait à partir du code de la tâche qui a été préalablement chargé par la procédure *InitGT*.
  - La tâche TN est lancée (*start_task*) en lui envoyant comme paramètre initial les identificateurs de MBN et MBC.

3.7.1.4 La tâche MODPER

La fonction de la tâche MODPER est de créer les modules permanents du système. Ces modules sont des modules de la couche CORTEX tels que GN et GM. Si cette couche CORTEX devait être étendue,

certains modules seraient lancés par cette tâche. Le source de la tâche *MODPER* est dans l'annexe H.2.

3.7.1.5 La tâche Relai

Cette tâche (annexe H.3) assure la création de la tâche TC d'un module OMPHALE. Cette fonction est réalisée à partir des primitives *load* et *unload* de l'agence programme ainsi que les primitives *create_task* et *start_task* de l'agence tâche. Cette tâche assure que seules les images binaires des modules nécessaires sont chargées en mémoire.

- La tâche TC gère la réentrance des codes de modules en mémoire. Pour ce faire, elle mémorise ces codes dans une liste chaînée.
- La *Creation* de la tâche TC
  La tâche *Relai* s'assure que le code de TC ne soit pas déjà chargé. Il faut pour cela parcourir la liste des codes et rechercher l'image mémoire:
  - Si la recherche est positive , une nouvelle tâche TC est crée à partir du code préalablement chargé. Le compteur d'utilisation du code est incrémenté de 1.
  - Si cette recherche n'aboutit pas, il faut donc charger le code de TC et créer une nouvelle entrée. Le compteur d'utilisation du code est initialisé à 1.

  La tâche peut ensuite être créée.
- La fonction *Destruction* est reçue lorsque l'exécution de TC est terminée (normalement ou pas). Le code de cette tâche TC est déchargé si la tâche détruite était la dernière à exécuter ce code.

3.7.2 La destruction de module OMPHALE

La destruction d'un module fait suite à plusieurs situations:

- L'initiative du module lui même. Cela correspond à une fin de programme. Cette fin peut être:
  - **normale**
    Le module décide de lui même de se terminer. Le module ferme la totalité des quais (au moyen des primitives *SetServ*, *PutServ* et *ResServ*). Quelque soient les messages arrivant au module, les services ne sont plus activables. Le noyau s'aperçoit alors que le module n'est pas réactivable (primitive *Attendre* de l'agence Routage).
  - **anormale**
    Une erreur de programmation a entrainé un trap fatal au module. Le traitement est pratiquement semblable à la fin de module sauf que les messages de la zone d'émission ne sont pas émis. Ce cas de terminaison est traité par la primitive *Failure* de l'agence Messages.
- D'autres modules:
  - La destruction du module peut se faire en détruisant l'ensemble des liens se trouvant dans le système. Le module meurt par isolement c'est à dire par absence de référence.
  - La destruction peut se faire explicitement. Le quai destruction est prévu à cet effet. Il contient les requêtes de destruction destinées au module.

3.8 Conclusion

La thèse de Jean-Marc Geib [GEI 89] constitue une sorte de cahier des charges. Pour conclure ce chapitre, nous résumerons ce qui a été fait et ce qu'il resterait à faire.

3.8.1 Ce qui est fait!

Les fonctionnalités suivantes ont été implantées:

- La communication asynchrone par messages entre modules. Les points suivants ont été respectés:
  - La **transparence** du réseau. Les modules communiquent d'une manière unifiée qu'ils appartiennent ou non au même site.
  - Le transport des messages distants par la station de transport de la couche CORTEX.
  - La gestion des zone d'émission et réception est conforme à la spécification.
- Le schéma original d'exécution dirigé par les messages reçus,
- Le mécanisme de nommage et protection par les liens,
- Le mécanisme de transfert de liens et donc d'édition de liens dynamiques,
- Une séparation des fonctions noyau et calcul a été proposée.
- L'architecture modulaire du SPS7 a été pleinement utilisée. Une version avec deux Modules de traitement a été testée (Un module de traitement sous SPIX et un module de traitement sous SPART).

3.8.2 Ce qu'il reste à faire...

Nous enumérerons dans cette section les fonctionnalités non implantées, ou les différences avec celles décrites dans le cahier des charges [GEI 89]. Nous rappelerons également les restrictions de l'implantation. Ces limitations pourraient être levées par une ré-écriture partielle du noyau.

- Pas de mécanisme d'attente limitée (time-out).
- Pas de priorité sur les services.
- La limitation sur la taille des environnements.
- La réutilisation des noms uniques.
- Un ramasse-miettes de modules OMPHALE plus évolué, qui contrôle les boucles de références de modules.

- La tolérance aux pannes n'a pas été étudiée dans le cadre du projet. Elle nécessite une révision importante de l'architecture OMPHALE.
- Le stockage des modules en mémoire secondaire n'a également pas été étudié dans le cadre du projet OMPHALE.
- L'écriture du noyau au dessus de SPART montre des limitations. Des mesures de temps sont difficilement obtenues. D'autre part, certains mécanismes SPART sont inadaptés à l'architecture OMPHALE. En particulier, la gestion mémoire, le chargement du code des modules sont internes au noyau SPART et donc implicitement au noyau NERF.

Chapitre 4
La programmation OMPHALE

Ce chapitre détaille des programmes OMPHALE. Le développement de ces programmes a pour but de:

1. Etudier une méthodologie de programmation adaptée à l'architecture OMPHALE et en particulier au noyau NERF.
2. Valider le noyau NERF en appelant ses primitives.

Un certain nombre de thèmes ont été abordés:

- **systèmes**
  Des modules systèmes ont été développés. Ces modules proposent des services "système" aux modules clients. Ce sont des modules serveurs. Ils composent la couche CORTEX du système OMPHALE. MINIX [TAN 87] propose une couche similaire composée de deux processus serveurs: Le *file system* et le *memory manager*.
- **structures de données**
  Nous avons étudié l'implantation de structures de données sur une architecture répartie. L'implantation de ces structures doit tenir compte de la répartition, pour rendre ensuite, les algorithmes portant sur ces structures les plus efficaces.
- **Synchronisation et coopération**
  Les problèmes de synchronisation et coopération demandent une parfaite maîtrise des outils de synchronisation. L'idée est de montrer que les problèmes de synchronisation et communication trouvent une solution simple et efficace sur l'architecture OMPHALE.

4.1 La méthodologie

On peut définir une méthologie de programmation comme un ensemble de règles à respecter pour la conception des programmes. De nombreux articles ont été publiés sur ce qui est (ou pas), bien de faire en matière de programmation. On peut citer par exemple le très célébre article de Dijkstra [DIJ68b]: "L'instruction GOTO est dangereuse". Une des méthodologies de programmation les plus connue est celle de la **programmation structurée**. Celle-ci a connu un grand succès avec l'apparition des langages de programmation tels que ALGOL [AFC 75] ou PASCAL [WIR 71].
Une nouvelle approche appelée **approche objet** est apparue au cours des années 1980. Elle connait un vif succès avec des langages tels que Smalltalk [GOL 83].
La méthodologie de programmation que nous proposons s'apparente à la méthodologie objet [GOO 88]. Elle consiste à:

- **Déterminer les modules du programme**
  B. Liskov [LIS 79] montre dans un article célèbre que la modularité est nécessaire pour le développement de programmes répartis. La modularité exprime naturellement la solution d'un problème à automatiser. L'analyse préalable consiste à établir les différentes entités du problème réel. L'idée va être ensuite d'associer à chaque entité du problème réel un module OMPHALE (objet) du programme. Le comportement du module OMPHALE doit calquer parfaitement avec celui de l'entité réelle.
  Notre propos sera illustré par l'exemple simplifié d'une entreprise de vente par correspondance. Cette entreprise est structurée de la manière suivante:

  - Un bureau des commandes
    La fonction de ce bureau est d'enregistrer les commandes des clients de la société.
  - Un bureau comptabilité
    Pour chaque commande enregistrée au bureau des commandes, le bureau comptabilité établit la facture de la commande. Ce bureau s'occupe également du règlement des clients.
  - Un bureau stock
    Le stock de l'entreprise est géré par ce bureau. A tout instant, ce bureau est capable de déterminer l'état du stock. Il assure les entrées dans le stock (livraisons fournisseurs) et les sorties (commandes clients).
  - Un bureau des expéditions
    A partir des factures communiquées par le bureau comptabilité, ce bureau assure la livraison des articles au client. Les articles de la facture sont sortis du stock et sont emballés en vue du transport.

  Pour chacun de ces bureaux, nous définirons un module qui simulera le fonctionnement du bureau.
  Les bureaux de l'entreprise travaillent en parallèle. La décomposition modulaire proposée par OMPHALE simule parfaitement la structuration du problème et son fonctionnement en parallèle. Les différents modules pourront s'exécuter en parallèle sur les différents sites de la configuration.

- **Services**
  Chaque entité du problème réel a des fonctions bien spécifiques à réaliser. Nous avons vu que les modules simulent parfaitement le fonctionnement de l'entité. Il parait donc naturel que le module réalise et propose les mêmes services que l'entité du problème. Pour chaque module, nous définirons alors un catalogue des services du module qu'on appelera l'**interface**. Pour exécuter un service, il est indispensable d'envoyer un message. Ce message contient les paramètres du service. Pour chaque service, nous établirons ses paramètres en définissant un format de message.
  Pour en revenir à notre exemple, le bureau commande envoie un message au bureau comptabilité à la fin de l'enregistrement de chaque commande. Ce message contient une commande enregistrée auprès d'un client. A partir de ce message (commande), le bureau comptabilité établit une facture. Les différents bureaux proposent les services suivants:

  - Le bureau commande propose deux services. Enregistrement et Annulation de commande. Ces services sont appelés par des modules clients que nous ne verrons pas ici.

- Le bureau comptabilité définit un service Facturer qui à partir d'un bon de commande issu du bureau commande constitue la facture associée à la commande. Un service Réglement est appelé par les modules clients et consiste à régler les factures.

- Le bureau stock gère donc le stock de la société. Il définit un service Sortir qui est utilisé par le service Expédition pour la constitution des commandes. Un service Entrer est appelé à l'arrivée de nouveaux articles en stock (Livraison par Fournisseur). Un service d'interrogation du stock est proposé. Il peut être consultable par le service Commande pour connaitre la disponibilité des articles.

- Le bureau Expédition a un service Expédier qui est activé par le service Comptabilité. A partir d'une facture, les artciles de la facture doivent être expédiés au client. Les articles doivent être sortis du stock (service Sortir).

- **Visibilité**
  Certaines entités du problème sont en relations étroites entre elles. On peut donc pour une application, établir un graphe des relations. Ce graphe répond à la question suivante: "Qui connait qui?". Si une entité a besoin de faire exécuter un service, elle doit nécessairement connaître l'entité qui réalise ce service. Il faut donc matérialiser ces dépendances au niveau de la programmation. Les relations entre les modules OMPHALE se font au travers des liens [GEI 89]. Un lien donne l'accès à certains services d'un module. L'ensemble des liens détenus par le module est mémorisé dans l'environnement du module. L'environnement du module précise dynamiquement la visibilité du module. Certains liens peuvent résider temporairement dans l'environnement. En particulier, dans le cadre d'un dialogue Question-Réponse, le module interrogé posséde temporairement un lien vers le module questionneur. Ce lien peut être détruit sitôt la réponse envoyée. On obtiendrait donc pour l'exemple de la société de vente par correspondance les dépendances suivantes:

  - Le module Commande a besoin de posséder un lien vers le module Comptabilité pour lui envoyer les commandes qu'il a reçues. Il doit connaitre le bureau stock s'il désire interroger le stock avant de passer une commande.

  - Le module Comptabilité reçoit du module Commande des messages contenant des commandes. A partir d'une commande, le module Comptabilité constitue la facture et l'envoie au module Expédition. Le module Comptabilité détient donc un lien vers le module Expédition.

  - Le module Expédition reçoit les factures du module comptabilité. Il demande au module Stock de sortir les différents articles du stock.Il doit donc posséder un lien vers le module Stock.

- L'application est souvent chapeautée par un module principal qui assure les fonctions suivantes:

  - Il effectue avec le module GM (créateur de module) les créations des différents modules de l'application.

  - Il communique aux différents modules les liens qu'ils ont vers les autres modules. Pendant l'exécution de l'application, les liens transitent dynamiquement entre les modules. OMPHALE est un système à édition de liens **dynamique** [COR 65].

4.2 Les langages de programmation utilisés

- Il était un temps où les systèmes d'exploitation étaient complétement écrits en langage d'assemblage. MULTICS [COR 65] [ORG 72] a été un des premiers systèmes à avoir été écrit en grande partie dans un langage évolué: Le PL/I [COR 69].
  Le langage C [KER 78] a ensuite été souvent utilisé pour le developpement de systèmes (UNIX [RIT 74], XINU [COM 83], MINIX [TAN 87]). Nous avons également utilisé le langage C pour le développement du noyau OMPHALE. En effet, BULL fournit une interface C pour les primitives du noyau SPART.

  Nous avons ensuite décidé ensuite de choisir un autre langage de programmation pour la programmation des modules OMPHALE (CORTEX et application). Le langage C n'est en effet pas toujours bien maîtrisé par les programmeurs. Des critiques [BRO 82] sont souvent émises à l'encontre du C:

  - L'**insécurité** du langage
    Les compilateurs C ne vérifient pas les paramètres effectifs des fonctions (types et nombre de paramètres). Les contrôles de types sont très succincts. Certains compilateurs proposent un mécanisme de prototypage des fonctions [BOR 87] [STR 86] qui augmentent les vérifications faites par le compilateur. Le langage C++ [STR 86] complète le langage C avec la notion de classe des langages objets.
  - Le **manque de lisibilité** du langage.
    Il est possible d'écrire des programmes très concis mais totalement illisibles. Certains ont la réflexion suivante: "Le C est plus proche d'un langage d'assemblage que d'un langage évolué!". La maintenance des programmes C n'est pas toujours évidente.
  - La **gestion mémoire**
    La gestion mémoire du programme est en partie à la charge du programmeur. Il dispose de plusieurs classes d'allocation (registres, allocation statique, allocation automatique) qu'il lui faut parfaitement maîtriser.

D'autre part, le langage C ne propose pas d'outils modulaires que sont les paquetages (ou modules) [DoD 83] [WIR 83].

4.2.1 MODULA-2

De nombreux langages répondaient à nos besoins mais n'étaient pas encore disponibles sur la SPS7. (ADA [DoD 83], C++ [STR 86]).
Notre choix s'est donc porté sur le langage MODULA-2 de N. Wirth [WIR 83] [BAU 84].
MODULA-2 reste dans la lignée du langage PASCAL [WIR 71] qui avait connu un vif succès, en particulier dans le milieu universitaire. Des critiques [KER 81] [KAI 83] avaient été émises à l'encontre de PASCAL. On reprochait en particulier à PASCAL de ne pas être adapté au développement de logiciels importants. D'autre part, PASCAL ne proposait pas d'outils pour l'écriture de systèmes. En particulier, il est difficile d'accéder aux interruptions de la machine et même d'accéder

individuellement aux bits d'un mot. N. Wirth a répondu en partie à ces critiques en proposant MODULA-2.

MODULA-2 propose donc en plus:

- Le **module**. Un programme MODULA-2 est structuré en modules compilables séparément. Chaque module posséde des listes explicites d'objets importés et exportés. Un objet importé (ou exporté) est une constante, un type, une variable ou bien une procédure. Les structures de données et les types peuvent être cachés. Ceci permet de représenter des *types abstraits de données* [BIS 86].
- Des outils pour le développement d'application systèmes.
  Un module *SYSTEM* contient des primitives de bas-niveau qui permettent la programmation des systèmes d'exploitation. Le module *SYSTEM* définit le concept de coroutine. La coroutine est un outil de décomposition de programme en processus dont un seul est actif à un instant donné [CON 63] [RAY 84]. Les coroutines MODULA-2 se synchronisent par des moniteurs [HOA 75].
- Certains détails syntaxiques ont été corrigés. En particulier, les structures de contrôles sont parenthésées par un END.

4.2.2 La compilation de modules OMPHALE

L'Université de Toulouse nous a cédé un compilateur MODULA-2 sous SPIX. L'Université de Toulouse utilise ce compilateur en enseignement mais aussi pour le développement du projet de recherche ACTOR [BET 85].
Ce compilateur MODULA-2 posséde une longue histoire que l'on peut rapidement conter.

- Ce compilateur a d'abord été développé à ZURICH par l'équipe de N. Wirth. (matériel C.D.C). Ce compilateur comprend 5 passes. Les passes 4 et 5 sont chargées de la génération du code machine.
- L'Université de Karlsruhe l'a ensuite porté sur VAX sous ULTRIX (UNIX BSD 4.2). Les travaux ont essentiellement porté sur les passes 4 et 5.
- L'université de Toulouse l'a enfin adapté aux systèmes SMX et SPIX des SM90 et SPS7.

Nous devions ensuite implanter le compilateur sur le système SPART. La phase de compilation proprement dite (production des objets) n'a pas été modifiée. Seule l'édition de liens a été corrigée. Ce travail consistait à reprendre les objets produits avec le compilateur MODULA-2 SPIX, et à y inclure le contexte d'exécution (run-time) SPART-NERF et les bibliothèques système SPART.

Dans un deuxième temps, nous avons écrit une interface du noyau NERF en MODULA-2. Le lecteur trouvera en annexe le module *Omphale* où l'on retrouve les déclarations de types et de primitives du noyau NERF. Tout ce qui est externe au noyau (couche CORTEX et applications) peut ainsi être programmé en MODULA-2.

4.2.3 L'architecture du module OMPHALE

Ce paragraphe décrit la structure du module OMPHALE au niveau du texte source MODULA-2. Un module OMPHALE s'implante à partir de deux modules MODULA-2.

- Un module d'interface
  Ce module MODULA-2 définit les messages ainsi que les primitives d'envoi et réception de messages.
- Un module de réalisation
  Ce module définit les services du module OMPHALE ainsi que la stratégie d'utilisation (contrôleur) de ces services.

Nous illustrerons cette structure par un module OMPHALE *Tampon*. La fonction de ce module est de mémoriser une valeur (service Placer) et de la restaurer (service Retirer). Ce module se comporte comme une position mémoire où seule une écriture suivie d'une lecture sont possibles.

4.2.3.1 Le module d'interface

Un module d'interface définit des formats de messages OMPHALE. Ce format de message s'exprime en terme d'enregistrement. Pour chaque format de message, le module d'interface définit deux procédures d'envoi et de réception de messages.
MODULA-2 est un langage typé. Il est obligatoire d'avoir une procédure d'envoi de messages par type de messages (idem pour la réception). La procédure d'emission est géneralement utilisée par les modules *Clients*. La procédure de réception est appelée par le module lui-même pour accéder au contenu du message reçu.

```
DEFINITION MODULE ITampon ;

FROM Omphale IMPORT ModuleId,ServiceId ;

EXPORT QUALIFIED
                PLACER, RETIRER,

                MESSPLACER, SendPlacer, GetPlacer,
                MESSRETIRER, SendRetirer, GetRetirer,
                SendReplyRetirer, GetReplyRetirer ;


CONST
     PLACER = 1 ;
     RETIRER = 2 ;

TYPE
     MESSPLACER = INTEGER ;

     MESSRETIRER = RECORD
```

```
                    Demandeur : ModuleId ;
                    QuaiDemandeur : ServiceId ;
                END ;


PROCEDURE SendPlacer (M : ModuleId; mess : MESSPLACER) ;

PROCEDURE GetPlacer (VAR mess : MESSPLACER) ;

PROCEDURE SendRetirer (M : ModuleId ; mess : MESSRETIRER) ;

PROCEDURE GetRetirer (VAR mess : MESSRETIRER) ;

PROCEDURE SendReplyRetirer (M : ModuleId ; Quai : ServiceId ;
                            mess : MESSPLACER) ;

PROCEDURE GetReplyRetirer (VAR mess : MESSREPLACER) ;
END ITampon.
```

Le module de définition MODULA-2 définit deux types de message. Le type *MESSPLACER* désigne le message reçu par le module *Tampon* pour le stockage d'une valeur. Le type *MESSRETIRER* définit le message d'interrogation du module Tampon. L'identité du module demandeur doit figurer dans ce message. Pour chaque type de message, le module définit les procédures d'envoi et de réception de message.

```
IMPLEMENTATION MODULE ITampon ;

FROM Omphale IMPORT ModuleId,ServiceId,MessageId,Null,
                    TRANSFERLINK,
                    CREATEMESSAGE,LASTMESSAGE,IDLINKRECEIVED ;

FROM Error   IMPORT Status ;
FROM SYSTEM  IMPORT ADR,TSIZE ;

VAR cr : Status ;
    IdM : MessageId ;

PROCEDURE SendPlacer (M : ModuleId; mess : MESSPLACER) ;
BEGIN
  cr := CREATEMESSAGE (M,PLACER,IdM,TSIZE(INTEGER),ADR(mess)) ;
END SendPlacer ;

PROCEDURE GetPlacer (VAR mess : MESSPLACER) ;
BEGIN
  LASTMESSAGE (ADR(mess)) ;
END GetPlacer ;

PROCEDURE SendRetirer (M : ModuleId ; mess : MESSRETIRER) ;
BEGIN
 cr := CREATEMESSAGE (M,RETIRER,IdM,TSIZE(ServiceId),
                      ADR(mess.QuaiDemandeur)) ;
```

```
 cr := TRANSFERLINK (mess.Demandeur,IdM) ;
END SendRetirer ;

PROCEDURE GetRetirer (VAR mess : MESSRETIRER) ;
BEGIN
 LASTMESSAGE (ADR(mess.QuaiDemandeur)) ;
 mess.Demandeur := Null ;
 cr := IDLINKRECEIVED (1,mess.Demandeur) ;
END GetRetirer ;

PROCEDURE SendReplyRetirer (M : ModuleId; Quai : ServiceId ;
                            mess : MESSPLACER) ;
BEGIN
  cr := CREATEMESSAGE (M,Quai,IdM,TSIZE(MESSPLACER),ADR(mess)) ;
END SendReplyRetirer ;

PROCEDURE GetReplyRetirer (VAR mess : MESSPLACER) ;
BEGIN
 LASTMESSAGE (ADR(mess)) ;
END GetReplyRetirer ;

BEGIN
END ITampon.
```

Dans le module d'implantation MODULA-2, nous retrouvons les procédures d'envoi de messages qui se définissent par:

- La création du message hors-liens. La primitive *CreateMessage* est appelée.
- Le transfert des liens dans le message. Ce transfert s'effectue par l'appel à la primitive *TransferLink*.

Les procédures de réception sont définies par:

- L'appel à la primitive *LastMessage* pour récupérer la partie du message hors-liens.
- La récupération des liens du message. La primitive *IdLinkReceived* permet de récupérer les liens du message.

4.2.3.2 Le module de réalisation

Les services du module OMPHALE, la stratégie d'exécution de ces services sont décrit dans le module de réalisation.
Le code d'un service est définit par une procédure MODULA-2. L'ensemble des procédures associées aux services du module OMPHALE est chargé dans le tableau *LesServices*. La primitive *Wait* retourne le numéro du service choisi par NERF. Le service est appelé en désignant l'entrée dans le tableau des services.

```
MODULE Tampon ;

FROM Omphale IMPORT Null, Reponse,
                    ModuleId, ServiceId, MessageId,
                    INITBUFFER,
                    DESTROYLINK, WAIT, PUTSERV, SETSERV,
                    TRANSFERLINK, DUPLICATELINK,
                    CREATEMESSAGE ;

FROM ITampon   IMPORT PLACER,RETIRER,
                      MESSPLACER,GetPlacer,
                      MESSRETIRER,GetRetirer,
                      SendReplyRetirer ;

FROM Error   IMPORT Status ;

VAR
    cr : Status ;
    IdM : MessageId ;
    LesServices : ARRAY ServiceId OF PROC ;

    Valeur : INTEGER ;

PROCEDURE Placer () ;
VAR m : MESSPLACER ;
BEGIN
  GetPlacer (m) ;
  Valeur := m ;
END Placer ;

PROCEDURE Retirer () ;
VAR m : MESSRETIRER ;
    m1 : MESSPLACER ;
BEGIN
    GetRetirer (m) ;
    m1 := Valeur ;
    SendReplyRetirer (m.Demandeur,m.QuaiDemandeur,m1) ;
END Retirer ;

BEGIN
     LesServices [PLACER] := Placer ;
     LesServices [RETIRER] := Retirer ;
     cr := INITBUFFER () ;
```

```
      SETSERV ({PLACER}) ;
      LesServices [WAIT ()] ;

      SETSERV ({RETIRER}) ;
      LesServices [WAIT ()] ;

      SETSERV ({}) ;
      LesServices [WAIT ()] ;
END Tampon.
```

- *Placer*
  Le contenu du message reçu est mémorisée dans la variable *Valeur.*
- *Retirer*
  Le contenu de la variable *Valeur* est retourné au module *Client.* Pour cela, le message reçu doit contenir un lien et un numéro de service chez le module *Client.*
- *Contrôleur du module*
  C'est la partie principale du module *Tampon.* C'est elle qui définit la politique d'acceptation du module. Sur notre exemple *Tampon,* le module accepte dans un premier temps la mémorisation d'une valeur, dans un second temps le retrait de cette valeur et enfin se termine en se suicidant *SetServ ({}).*

Cette description montre une certaine lourdeur dans la phase de développement. Jean-Marie Place [PLA 89] a développé un pré-processeur d'un langage OMPHALE en MODULA-2. En particulier, le module d'interface (messages et procédures sur ces messages) est généré automatiquement par le pré-processeur.
Pour simplifier la présentation, nous décrirons nos exemples en ne décrivant que les services et le contrôleur des modules de réalisation. Les modules d'interface ne seront pas décrits pas dans cette thèse.

4.3 Les modules du CORTEX

La couche CORTEX du système OMPHALE se compose d'un ensemble de modules systèmes. Ces modules proposent des services "systèmes" de haut-niveau tels que la gestion mémoire, le nommage symbolique et la création dynamique de modules. Ceci a pour conséquence de dégager du noyau des fonctionalités. MINIX [TAN 87] propose une structuration similaire avec deux processus serveurs: Le *File Server* et le *Memory Manager*.

Les modules du CORTEX fonctionnent suivant le modèle **Client-Serveur**. Chaque module du CORTEX va proposer des services "systèmes". A chaque service correspondra une file d'attente de la zone de réception. Pour être client d'un module du CORTEX, les modules devront posséder un lien vers ces modules du CORTEX. Pour utiliser un service d'un module du CORTEX, le lien devra être utilisable pour ce service.

4.3.1 Le gestionnaire de noms GN

Ce module définit un mécanisme de nommage symbolique des modules. Un nom symbolique est une chaîne de caractères. Pour chaque nom symbolique, le GN associe un lien. Le gestionnaire de noms gère un certain nombre d'associations (nom symbolique,lien). GN propose trois services.

- **EnregNameLink**
  Création d'une association d'un nom symbolique et d'un lien.
- **AskLink**
  Recherche du lien correspondant à un nom symbolique.
- **DeleteNameLink**
  Destruction d'une association nom symbolique-lien.

4.3.1.1 EnregNameLink

Pour utiliser ce service, le module client doit envoyer un message du type *MESSENREG*. Ce message contient:

- un nom symbolique (*Name*),
- un lien (*IdLien*).
  Les modules ne manipulent pas directement les liens. *IdLien* désigne un lien de l'environnement du module GN.

GN conserve les associations dans une table. Le service *EnregNameLink* ajoute une entrée dans cette table. Une entrée de la table comprend un nom symbolique et un lien attaché à ce nom.

```
PROCEDURE EnregNameLink ;
VAR
   m : MESSENREG ;
   x : INTEGER ;
BEGIN
    GetEnreg (m) ;

    x := PlaceVide () ;
    IF x <> NotFound
       THEN
           Table [x].Occupe := TRUE ;
           Table [x].ModuleName := m.Name ;
           Table [x].IdLien := m.Lien ;
       ELSE
           (* appel aux procedures d'entrees sorties MODULA-2 *)
           (* pour afficher erreur limitation environnement module *)
           (* Il faudrait envoyer un message au module OMPHALE d'entree *)
           (* sortie *)
           WriteString ("GN sature") ;
           WriteLn ;
    END ;
END EnregNameLink ;
```

La version actuelle limite le nombre d'entrées de l'environnement des modules. L'environnement du module GN est également limité. La conséquence est que GN ne peut gérer qu'un nombre limité de noms symboliques. D'où l'utilisation d'une table. La fonction *PaceVide* recherche dans la table des associations une entrée vide. Elle retourne *NotFound* si la table est saturée.

Le nombre de noms symboliques est donc limité au nombre d'entrées de l'environnement. Nous pouvons remédier à ce problème en proposant un chaînage de module GN. Lorsque la table des associations est pleine, le module GN crée un nouveau module GN et lui envoie la demande d'enregistrement. Les modules GN sont alors chaînés entre eux.

Si l'environnement n'était pas de taille limitée, on pourrait alors gérer au niveau du module GN une liste chaînée d'associations.

4.3.1.2 AskLink

Ce service vise à obtenir une copie du lien associé à un nom symbolique. Un message *MESSASKLINK* est envoyé au module GN. Ce message comprend outre le nom symbolique, l'identité du module *Client* (Lien et un numéro de service).

```
PROCEDURE AskLink ;
VAR x : INTEGER ;
    m    : MESSASKLINK ;
    mr   : MESSREPLYASK ;
BEGIN
```

```
    GetAsk (m) ;
    x := RechercheName (m.Name) ;
    mr.Trouve := x <> NotFound ;
    IF mr.Trouve
      THEN
         cr := DUPLICATELINK (Table [x].IdLien,mr.Lien) ;
      ELSE
         mr.Lien := Null ;
    END ;
    SendReply (m.Demandeur,m.Quai,mr) ;
END AskLink ;
```

La fonction *RechercheName* recherche le nom symbolique *m.Name* dans la table. Elle renvoie un indice dans cette table. Si cet indice est valide, le lien est dupliqué; Il est donc nécessaire d'enregistrer dans le gestionnaire de noms, des liens dont la duplication est autorisée.

4.3.1.3 DeleteNameLink

Ce service doit détruire une association lien nom symbolique.

```
PROCEDURE DeleteNameLink ;

VAR
   x : INTEGER ;
   m : MESSDELETE ;

BEGIN
    GetDelete (m) ;

    x := RechercheName (m.Name) ;
    IF x <> NotFound
       THEN
           cr := DESTROYLINK (Table [x].IdLien) ;
           Table [x].Occupe := TRUE ;
    END ;
END DeleteNameLink ;
```

Le nom symolique *Name* se trouve dans le message reçu. Il est recherché dans la table des noms symboliques. Si ce nom est retrouvé, l'entrée de la table est de nouveau disponible et le lien est détruit.

4.3.1.4 Commentaires

Le module GN se comporte comme un module *Serveur*. Pour chaque message reçu, il exécute le service associé au message. Le contrôle du module s'effectue de la manière suivante.

```
SETSERV ({ASKLINK,ENREGNAMELINK,DELETENAMELINK}) ;
LOOP
    LesServices [WAIT ()] ;
END ;
```

Les services sont validés par la primitive *SETSERV*. Le module rentre ensuite dans une boucle infinie de lecture de messages.

A sa création, les modules OMPHALE possédent un lien vers le gestionnaire de noms de leur site. La station de transport (fonction réseau) est cliente du module GN. A chaque nouveau site connecté, elle effectue un enregistrement du module GN de ce nouveau site. (nom site, lien vers le GN de ce site)

4.3.2 Le gestionnaire de modules GM

Le module GM est responsable de la création de module. Il propose donc un certain nombre de services de création aux modules clients. Pour créer un module, le module client (père) doit s'adresser au gestionnaire de module GM. Il lui envoie un message de création avec son identité ainsi que le nom (prototype ou fichier exécutable) du module à créer. GM assure la création du module "fils".
Il existe quatre modes de création:

- **Création locale**
  Le module GN crée le module fils sur le même site que lui.
- **Création anonyme**
  Le module *Client* demande à créer un module sur un des sites de la configuration. Le module GM LOCAL choisit un site. La requête de création sera transmise au module GM de ce site.
- **Création sur site**
  Le module *Client* impose un site pour l'exécution de son fils. La requête de création contient, en plus, le nom du site où doit se faire la création.
- **Création régulante**
  Le module *Client* demande à ce que le module créé le soit sur le site le moins chargé. Ce service permet d'une certaine manière de réguler la charge des différents sites.

D'autre part, le système de transport (module station de transport de la couche CORTEX) communique au module GM les liens vers les autres modules GM des sites de la configuration.

4.3.2.1 Création locale

```
PROCEDURE NewModuleLocal ;
VAR
   mm : MESSNEWMODULE ;
   mr : MESSREPLYNEWMODULE ;
BEGIN
   GetNewModule (mm) ;
   mr.Lien :- Null ;
   cr :- CREATEMODULE (mm.Proto,mr.Lien) ;
   mr.Trouve :- cr - GOOD ;
   SendReplyNew (mm.Demandeur,mm.Quai,mr) ;
END NewModuleLocal;
```

- Le message reçu contient le nom du fichier exécutable (*mm.Proto*) correspondant au code du module à créer. Le message contient également les références du module *Client* (lien *mm.Demandeur* et service *mm.Quai* nécessaires pour la réponse).
- La primitive *CreateModule* du noyau NERF est exécutée. Elle retourne à GM un lien vers le module créé (*mr.Lien*). Sur cette implantation du noyau NERF, la primitive *CreateModule* charge le binaire et alloue les ressources mémoires du module. Ces fonctions sont normale-

ment à la charge de modules systèmes du CORTEX. Le noyau NERF est fortement dépendant de SPART.

- Le lien vers le module crée est placé dans le message de réponse au module *Client*. Le message de réponse est envoyé par la procédure *SendReplyNew*.

4.3.2.2 Création sur site nommé

```
PROCEDURE NewModuleSite ;
VAR
   m : MESSNEWMODULESITE ;
   mr : MESSREPLYNEWMODULE ;
   me : MESSNEWMODULE ;
   i  : INTEGER ;
BEGIN
   GetNewModuleSite (m) ;
   i := RechercheSite (m.Site) ;
   IF i = NotFound
      THEN
         mr.Trouve := FLASE ;
         SendReplyNew (m.Demandeur,m.Quai,mr) ;
      ELSE
         me.Demandeur := m.Demandeur ;
         me.Quai := m.Quai ;
         me.Proto := m.Proto ;
         SendNewModuleLocal (LesSites [i].Lien,me) ;
   END ;
END NewModuleSite ;
```

- GM posséde une table des sites connectés (*LesSites*). Chaque entrée de la table comprend le nom du site et un lien vers le module GM de ce site.
- Le module *Client* demande une création de module sur le nom de *Site* qui se trouve dans le message. Le *Site* est recherché (fonction *RechercheSite*) dans la table des sites connectés. Si cette recherche aboutit, le message est transmis au gestionnaire de module de ce site. C'est ce getionnaire de module qui enverra la réponse au module *Client* (long-retour [STR 81]).
- Si la recherche dans la table n'aboutit pas, une réponse est retournée directement au module *Client*.

4.3.2.3 Création sur un site quelconque

Le module *Client* (père) demande une création de module mais laisse au module GM le soin de choisir le site de création.

```
PROCEDURE NewModule ;
VAR
   mm : MESSNEWMODULE ;
```

```
   i  : INTEGER ;
BEGIN
   GetNewModule (mm) ;
   i := ChoixSite () ;
   SendNewModuleLocal (LesSites [i].Lien,mm) ;
END NewModule ;
```

- La procédure choisit un site (fonction *ChoixSite*) parmi l'ensemble des sites connus de la table *LesSites*. Cette procédure répartit équitablement ses choix.
- La requête est émise vers le module GM du site sélectionné. Si le site sélectionné est le site courant, le module GM s'envoie alors un message à lui-même.

4.3.2.4 Création sur le site le moins chargé

```
PROCEDURE NewModuleSiteLessLoaded ;
VAR
   mm : MESSNEWMODULE ;
   i  : INTEGER ;
BEGIN
   GetNewModule (mm) ;
   i := ChoixSiteMoinsCharge () ;
   SendNewModuleLocal (LesSites [i].Lien,mm) ;
END NewModuleSiteLessLoaded ;
```

- La procédure *ChoixSiteMoinsCharge* sélectionne dans la table des sites, le site qui est le moins chargé. Nous exprimons la charge en terme de nombre de modules qui sont actifs.
- La requête est transmise au module GM du site sélectionné.

4.3.3 La station de transport

La station de transport est une module système de la couche CORTEX. Elle assure le transfert des messages vers les modules distants (s'exécutant sur un autre site). L'agence SPART-ETHERNET n'a jamais fonctionné, ce qui nous a obligé de déporter la station de transport sous UNIX. La station de transport n'est donc pas interne au noyau NERF, mais déportée dans un module de la couche CORTEX s'exécutant sous UNIX.
UNIX et le noyau SPART permettent la communication et la synchronisation entre processus UNIX et SPART. Les sémaphores, segments mémoires, boîtes aux lettres UNIX (système V) et SPART sont compatibles. Ce partage n'est possible que si l'objet SPART est placé dans la catalogue global du système.

La station de transport a deux fonctions principales.

1. Le transfert des messages OMPHALE entre modules distants (émétteurs et destinataires résidant sur des sites différents).
2. La gestion des sites connectés. Nous verrons dans un premier temps les informations qu'un site conserve sur les autres sites présents. Nous pourrons ainsi en déduire les actions à exécuter à l'arrivée d'un nouveau site sur le réseau.

4.3.3.1 Transfert de messages

L'appel à la primitive *Wait* du noyau marque la fin de la phase d'activité du module OMPHALE. Le noyau va extraire les messages de la zone d'émission du module. Si le message est destiné à un module du site, le message est directement placé dans la boîte aux lettres MBN du module destinataire. Si tel n'est pas le cas, le message est placé dans une boîte aux lettres *MBDIST* qui contient les messages destinés aux modules distants (n'appartenant pas au même site). La fonction *EnvoiST* assure l'émission du message vers la station de transport. Le noyau NERF conserve le nom de la boîte aux lettres *MBDIST* de la station de transport. Les messages de cette boîte aux lettres sont receptionnés par un processus UNIX qu'on appelle **ST_EMETTEUR**. Pour chaque message de cette boîte aux lettres *MBDIST*, le processus **ST_EMETTEUR** le transmet sur le réseau vers le site du module destinataire. Le transfert sur le réseau se fait en utilisant les SOCKET TCP/IP [SUN 86]. Cette transmission se fait par copie du message OMPHALE sur une "socket" de type datagram simple (en mode non fiabilisé de transfert). Sitôt le message émis, le processus **ST_EMETTEUR** se met en attente d'un nouveau message sur la boîte aux lettres *MBDIST*.

Un processus UNIX **ST_RECEPTEUR** se met en attente de messages sur la socket. Dès qu'un message arrive du réseau, un message OMPHALE est créé à partir du contenu reçu sur la socket. Ce message est transmis à une tâche SPART qu'on appelle **INTER**. Cette tâche place les messages OMPHALE reçus du processus **ST_RECEPTEUR** dans la boîte aux lettres MBN du module destinataire du message.

4.3.3.2 Gestion des sites connectés

Les modules du CORTEX (GN et GM) ont besoin de posséder des liens vers les modules du CORTEX d'autres sites.

Tout site émet périodiquement un message d'identification sur le réseau (toutes les 30 secondes). Il émet ce message sur une socket datagram en diffusion. Ce message comprend le numéro INTERNET du site ainsi que des informations sur le site. Un processus UNIX *ST_ESPION* se met à l'écoute du réseau et des messages d'identification. A la réception d'un message d'identification, le processus *ST_ESPION* scrute sa table des sites connectés. Si le message reçu est le premier provenant du site, le site est ajouté dans la table et un message est émis vers GM et GN pour leur indiquer l'arrivée d'un nouveau site. Ce message contient le nom du site ainsi qu'un lien vers le module GN ou GM du site. On parcourt également cette table pour identifier les sites "morts" dont on n'a plus aucune nouvelle. Lorsqu'on détecte un site mort, on envoie un message à GN et à GM pour les prévenir que le site n'est plus accessible. Les liens vers les modules du site mort sont alors détruits.

4.4 Le module Pile

Un des objectifs de cette section est d'illuster le fonctionnement des mécanismes OMPHALE (Envoi de messages, Transfert de Liens, Création de modules) par des exemples.
La Pile est une structure de donnée abondamment décrite dans la littérature informatique [PAI 77] [MEY 80] [GAU 87]. La pile est un cas particulier de liste où les ajouts et les suppressions ne se font qu'à une seule extrémité appelée le *sommet de pile*. La structure Pile est du type L.I.F.O. (Last-In-First-Out). Cette structure est relativement simple. Cette structure est très connue et sert souvent d'illustration dans les manuels de programmation [DoD 83]. Cette structure de donnée fait partie de la vie courante (pile d'assiettes, pile de dossiers en attente de traitement).

4.4.1 Une pile OMPHALE

Nous réaliserons une pile OMPHALE à partir d'un module dit **Serveur**. Ce module propose les services suivants:

- **Empiler**
  Ce service se doit d'empiler une valeur se trouvant dans un message reçu à l'activation du service.
- **Depiler**
  Le module *Pile* se doit de retourner au module *Client* la valeur qui se trouvait au sommet de la pile.
- **PilePleine et PileVide**
  Le module *Pile* retourne au module *Client* un booléen indiquant si la pile est pleine ou vide suivant le service.

Les modules *Clients* n'accédent à la pile que par les differents services proposés sur cette pile. L'implantation de la pile est cachée dans l'implantation du module. C'est le principe de l'abstraction de type [BIS 86]. Nous avons représenté la pile à l'aide d'un tableau ce qui facilitera la présentation.

4.4.1.1 Empiler

```
CONST
    Taille = 100 ;
       .
       .
       .
VAR
    (* Variable locale au module Pile *)
    Sommet : INTEGER
    Zone   : ARRAY [1..Taille] OF INTEGER ;
       .
       .
```

```
PROCEDURE Empiler () ;
VAR m : MESSEMPILER ;
BEGIN
   GetEmpiler (m) ;
   INC (Sommet) ;
   Zone [Sommet] := m ;
   PUTSERV ({DEPILER}) ;
   IF Sommet = Taille
      THEN
         RESSERV ({EMPILER}) ;
   END ;
END Empiler ;
```

- Le message *m* contient la valeur à empiler. Cette valeur est rangée dans le tableau *Zone*. La valeur de *Sommet* est incrémentée de 1.
- Lorsque le service *Empiler* est exécuté, le service *Depiler* est accepté.
- Lorsque la *Zone* est saturée, le module n'accepte plus d'empiler et ferme donc le service *Empiler*.

4.4.1.2 Depiler

```
PROCEDURE Depiler () ;
VAR me : MESSINTER ;
    ms : MESSREPLYDEPILER ;
BEGIN
     GetInter (me) ;
     ms := Zone [Sommet] ;
     DEC (Sommet) ;
     SendReplyDepiler (me.Demandeur,me.Quai,ms) ;
     cr := DESTROYLINK (me.Demandeur) ;
     PUTSERV ({EMPILER}) ;
     IF Sommet = 0
        THEN
           RESSERV ({DEPILER}) ;
     END ;
END Depiler ;
```

L'identité du module *Client* (Un lien) se trouve dans le message reçu sur le service *Depiler*. Le service *Depiler* se termine par l'envoi d'un message contenant cette valeur.

Lorsque la pile est vide (*Sommet = 0*), le service *Depiler* est fermé.

4.4.1.3 Destruction de la pile

Pour détruire une pile, le module *Client* peut envoyer un message au service *DetruirePile*. Le module Pile se termine alors de lui-même.

```
PROCEDURE DetruirePile () ;
VAR me : MESSDELETE ;
BEGIN
   GetDelete (me) ;
   SETSERV ({}) ;
END DetruirePile ;
```

La destruction peut également être réalisée par un envoi de message sur le service Destruction du module. Tout module posséde un service Destruction. L'arrivée d'un message sur ce service entraine la destruction du module par le noyau.

Une autre manière de réaliser la destruction de la pile est d'isoler le module Pile. Si l'ensemble des liens référençant le module Pile se réduit à l'ensemble vide, la pile devient inacessible et le noyau détruit alors le module.

4.4.1.4 Commentaires

La pile OMPHALE que nous venons de présenter est donc partageable entre plusieurs module clients. Au début de son exécution, la pile n'est connue que de son module père (celui qui en a demandé la création au module GM). Il est alors le seul à posséder un lien vers ce module. Le module père peut dupliquer (*DuplicateLink*) ce lien pour ensuite le communiquer (*TransferLink*) à d'autres modules. Le module père pourra leur restreindre l'accès (*SetNoUse*) à cette pile en leur interdisant certains services. On peut imaginer des modules ne possédant pas le droit d'empiler. Le service empiler n'est pas alors validé sur le lien.

```
SETSERV ({DELETE, EMPILER}) ;

LOOP
  LesServices [WAIT ()] ;
END ;
```

Au début du module, la pile est vide et on n'accepte donc uniquement les services *Destruction* et *Empiler*.

4.4.2 Une deuxième pile OMPHALE

Cette deuxième implantation exploite mieux les possibilités de modularité de l'architecture. Le module **serveur** de Pile présente toujours la même interface. L'implantation des services est différente (abstraction). A chaque élément de la pile, nous associons un module *Elem* qui va mémoriser la valeur de l'élément de la pile. Chaque module *Elem* conserve un lien vers le module *Elem* suivant de la pile. Le module *Pile* conserve un lien vers le premier module *Elem* au sommet de la pile.

- L'opération *Empiler* consiste à créer un nouveau module de type *Elem* et à lui envoyer la valeur à mémoriser ainsi que le lien vers le module suivant.
- Le service *Depiler* consiste à émettre un message vers le module *Elem* se trouvant en tête. Ce message contient les références du module **Client**. C'est le module *Elem* qui va retourner la valeur au module *Client* (technique du long-retour [STR 81]). Le module *Elem* doit envoyer au module *Pile* le lien vers le module suivant qui sera ainsi le nouveau sommet de la pile. Le module *Elem* meurt ensuite.

4.4.2.1 Empiler

```
PROCEDURE Empiler () ;
VAR m : MESSEMPILER ;
    mn : MESSNEWMODULE ;
    mp : MESSPLACER ;
BEGIN
     GetEmpiler (m) ;

     mn.Demandeur := Myself ;
     mn.Quai := NEWELEM ;
     mn.Proto := 'Elem' ;
     SendNewModule (ServeurModule,mn) ;

     mp.Suivant := Sommet ;
     mp.Valeur := m ;

     SETSERV ([NEWELEM]) ;
     LesServices [WAIT ()] ;
     (* Creation d'un nouvel element *)

     SendPlacer (Sommet,mp) ;
END Empiler ;
```

Le service *Empiler* nécessite d'abord la création d'un module *Elem*. Un message est donc émis vers le module GM du CORTEX. Le module *Pile* attend une réponse du module GM qui arrive sur le service *NewElem*. Le point original de ce service est d'avoir la possibilité d'exécuter la primitive *Wait* à l'intérieur du service *Empiler*. Le service *NewElem* est appelée à partir de *Empiler*.

```
PROCEDURE NewElem () ;
VAR
   mr : MESSREPLYNEWMODULE ;
 BEGIN
      GetReplyNew (mr) ;
      IF mr.Trouve
         THEN
             (* memorise le nouveau sommet *)
             Sommet := mr.Lien ;
      END ;
      SETSERV ({EMPILER,DEPILER,DELETE}) ;
END NewElem ;
```

Dès que le module *Elem* est crée, un message lui est envoyé (*SendPlacer*) avec la valeur à empiler et un lien vers le module *Elem* suivant dans la pile.

4.4.2.2 Depiler

```
PROCEDURE Depiler () ;
VAR me : MESSINTER ;
    ms : MESSREPLYDEPILER ;
    mr : MESSRETIRER ;
    MySelfCopie : ModuleId ;
BEGIN
     GetInter (me) ;
     mr.Demandeur := me.Demandeur ;
     cr := DUPLICATELINK (Myself,MySelfCopie) ;
     mr.ServeurPile := MySelfCopie ;
     mr.QuaiDemandeur := me.Quai ;
     mr.QuaiServeur := NEWTETE ;
     SendRetirer (Sommet,mr) ;
     SETSERV ({NEWTETE}) ;
     LesServices [WAIT ()] ;
END Depiler ;
```

L'exécution du service *Depiler* fait suite à la réception d'un message provenant d'un module *Client*. Un message est envoyé au module *Elem* au sommet de la pile pour lui indiquer que l'on dépile. Le module *Elem* renvoie au module *Client* sa valeur (technique du long-retour [STR 81]). Le module *Elem* retourne ensuite au module *Pile* la valeur de l'élément suivant. Ce sera le nouveau Sommet. Ce lien vers le nouveau sommet est retourné sur le service *NewTete*.

```
PROCEDURE NewTete () ;
VAR
   m : MESSNEWTETE ;
BEGIN
   GetNewTete (m) ;
   Sommet := m.Tete ;
```

```
   IF Sommet = Null
     THEN
         SETSERV ({EMPILER,DELETE}) ;
     ELSE
         SETSERV ({EMPILER,DEPILER,DELETE}) ;
     END ;
END NewTete ;
```

L'exécution du service Dépiler par le module *Pile* se fait donc en collaboration avec le module *Elem.* On dira alors que le module *Pile* sous-traite le service par un autre module.
Dans le service *NewTete*, la variable *Sommet* est mise à jour. Si elle est à *Null*, le service *Depiler* n'est plus validé.

4.4.2.3 Le module Element

Le module élément se comporte comme à un buffer à une case:

- Le contrôleur du module accepte dans un premier temps de *Placer* une valeur dans la case. Cette valeur est restituée au premier appel du service *Retirer*.

```
     Suivant := Null ;

     SETSERV ({PLACER}) ;
     LesServices [WAIT ()] ;

     SETSERV ({RETIRER}) ;
     LesServices [WAIT ()] ;

     SETSERV ({}) ;
     LesServices [WAIT ()] ;
```

-
```
PROCEDURE Placer () ;
VAR m : MESSPLACER ;
BEGIN
  GetPlacer (m) ;
  Valeur := m.Valeur ;
  Suivant := m.Suivant ;
END Placer ;
```

  Pour ce service, le module *Element* mémorise la valeur ainsi qu'un lien vers le module Element suivant dans la pile.

-
```
PROCEDURE Retirer () ;
VAR m : MESSRETIRER ;
    m1 : MESSREPLYRETIRER ;
    m2 : MESSNEWTETE ;
BEGIN
```

```
    GetRetirer (m) ;
    m1 := Valeur ;
    SendReplyRetirer (m.Demandeur,m.QuaiDemandeur,m1) ;
    m2.Tete := Suivant ;
    SendNewTete (m.ServeurPile,m.QuaiServeur,m2) ;
END Retirer ;
```

La valeur mémorisée est envoyée au module client (qui a demandé à dépiler). La nouvelle tête est envoyée au module serveur Pile.

- On peut bien sûr douter de l'efficacité de cette solution. En effet, un module *Elem* existe pour chaque élement de pile. Deux tâches SPART sont nécessaires par élément de pile. C'est beaucoup! Mais cette solution est conceptuellement jolie...
- Chaque valeur empilée nécessite une création de module. On peut imaginer une solution où le module Pile gère un pool de modules *Elem* libres. L'appel du service *Depiler* aurait pour conséquence de placer un nouveau module *Elem* dans le pool des modules libres. L'appel du service Empiler puiserait dans ce pool un module libre. Si le pool est vide, cela aurait pour conséquence d'entrainer une nouvelle création de module *Elem.*
- On remarquera que l'accès à la pile est toujours le même, mais que l'implantation a totalement changé. C'est l'**abstraction** [BIS 86]. Pour le service dépiler, le module **Client** reçoit la valeur dépilée sans s'apercevoir qu'elle ne provient pas du module à qui il en a fait la demande.
- Lorsque le module client meurt, si il est le seul à connnaitre le module Pile, cela va entrainer la mort du module Pile. La mort de ce module entraine la mort du premier élement de pile. Et ainsi de suite jusqu'a la mort du module *Elem* se trouvant en fond de pile. C'est une véritable épidémie...
- Les éléments de la pile sont répartis sur le réseau.

4.5 Les philosophes et les spaghettis

4.5.1 Le problème

Ce problème a été présenté et résolu par E.W Dijkstra [DIJ 71]. Depuis, il est considéré comme un exercice classique de synchronisation, non par son intérêt pratique mais comme un bon exemple de problème de concurrence.

Cinq Philosophes sont invités à une soirée. Durant la soirée, le philosophe est dans l'un des deux états suivants:

- **Manger**
  Il mange, et des spaghettis (très certainement à la bolognaise) sont au menu. Le philosophe sait que pour manger des spaghettis, il a besoin de deux fourchettes.
- **Penser**
  Il réflechit aux problèmes philosophiques d'actualité. Pour penser, un philosophe n'a pas besoin de ses deux fourchettes.

En fait le philosophe ne mange pas en une seule fois. Son repas se compose d'un certain nombre de petites sessions où il mange. Quand il a terminé une session manger, il se remet à penser. Quand il a fini de penser, il recommence à manger et ainsi de suite...

Les cinq philosophes sont assis autour d'une table circulaire. Au centre de cette table se trouve un magnifique plat de spaghettis. Devant chaque philosophe, se trouve une assiette entourée de deux fourchettes. Chaque fourchette est succeptible d'être utilisée par deux philosophes voisins (assis l'un à côté de l'autre)

Quand un philosophe pense, il n'utilise pas ses deux fourchettes. Il les laisse à la disposition de ses deux voisins.

Quand un philosophe désire manger, il doit se saisir des deux fourchettes. Il ne peut prendre qu'une fourchette posée sur la table. Il ne peut donc pas prendre une fourchette dans la main d'un de ses voisins.

Quand il a fini de manger, il repose les deux fourchettes sur la table.

La solution proposée par Dijkstra [DIJ 71] était construite autour des sémaphores.

```
VAR
   Four : ARRAY [0..4] OF SEMAPHORE;
   (* Valeur du semaphore est initialise a 1 *)

LOOP
```

```
        ...
           penser
        ...

      P (Four [i]) ;
      P (Four [i+1 MOD 5]) ;

        ...
           manger
        ...

      V (Four [i]) ;
      V (Four [i+1 MOD 5]) ;
END ;
```

4.5.2 Les modules

Une solution distribuée est proposée par T.A. Cargill [CAR 82]. Elle est distribuée en ce sens que la synchronisation et la communication ne se limite qu'à deux philosophes voisins. Il n'y a pas mécanisme centralisé qui dénote l'état global des philosophes.

Le comportement de chaque philosophe se modélise par un processus (module OMPHALE). Les fourchettes sont également modélisées par des modules. Les fourchettes et les modules possédent un numéro allant de 1 à *nb* (*nb* étant le nombre de philosophes invités à la soirée). Pour manger, le Philosophe doit se saisir de ses deux fourchettes. Pour se saisir d'une fourchette, le philosophe envoie un message à la fourchette pour lui indiquer qu'il désire l'utiliser. Le philosophe reste bloqué tant que la fourchette ne lui répond pas. Le philosophe ne commence à manger que lorsqu'il a obtenu une réponse de ses deux fourchettes. Lorsqu'il a terminé de manger, le philosophe envoie un message à ses deux fourchettes pour leur indiquer qu'elles sont maitenant libres.

Le module maître d'hôtel initialise l'application. C'est lui qui crée les modules *Philosophes* et *Fourchettes* en collaboration avec le module GM. Il envoie ensuite à chaque philosophe son numéro ainsi que des liens vers ses fourchettes. La visibilité du philosophe n'est ainsi limitée qu'à ses deux seules fourchettes.

Il y a deux classes de philosophes. Les philosophes avec un numéro pair et ceux avec un numéro impair. Pour manger, les philosophes avec un numéro pair se saisiront d'abord de la fourchette gauche puis de la droite. Les philosophes impairs font l'inverse.

T.A. Cargill montre que sa solution est robuste, c'est à dire que la défaillance d'un philosophe (utilisant une fourchette) ne bloque pas l'ensemble du système.

4.5.2.1 La fourchette

Chaque fourchette est modélisée par un module OMPHALE. Ce module comprend deux services:

- service *Prendre*
  Lorsque ce service est accepté, la fourchette passe dans la main d'un des deux philosophes succeptibles de l'utiliser.

```
PROCEDURE Prendre () ;
VAR
    m : MESSPRENDRE ;
    mr : MESSREPLYPRENDRE ;
    Status : cr ;
BEGIN
    GetPrendre (m) ;
    SendReplyPrendre (m.Demandeur,m.Quai,mr) ;
```

```
    cr := DESTROYLINK (m.Demandeur) ;
END Prendre ;
```

Le message reçu contient un lien vers le philosophe demandeur. Une réponse est envoyée au module *Philosophe* par la procédure *SendReplyPrendre*. Le lien ayant servi à la réponse au philosophe est ensuite détruit.

- service *Poser*
  Le philosophe qui utilisait la fourchette la repose après avoir terminé une session "manger". Le service *Prendre* peut de nouveau être accepté.

```
PROCEDURE Poser () ;
BEGIN
END Poser ;
```

- Le module *Fourchette* contrôle son utilisation en alternant les services *Prendre* et *Poser*. Le contrôle a donc la forme suivante:

```
    LOOP
        SETSERV ({PRENDRE}) ;
        LesServices [WAIT ()] ;

        SETSERV ({POSER}) ;
        LesServices [WAIT ()] ;
    END ;
```

4.5.2.2 Le philosophe

Le comportement du philosophe est modélisé par le module *Philosophe*. Pour manger, il essaiera de s'approprier les fourchettes. L'obtention d'une fourchette exige un envoi de message au module fourchette correspondant.

- *service d'initialisation*
  Le message d'initialisation contient:
  - Un lien vers le module *Observateur*. Ce module est chargé des opérations entrées-sorties et de l'affichage de l'état des philosophes.
  - Deux liens vers ses deux fourchettes. Le module *Philosophe* ne connait que deux seules fourchettes et ne peut donc ainsi subtiliser des fourchettes auxquelles il n'a pas droit.
  - Un numéro *Num*. Il permet de déterminer l'ordre dans lequel les philosophes saisissent les fourchettes. Les philosophes avec un numéro impair se saisissent d'abord de la fourchette gauche puis ensuite la droite. (Les pairs font l'inverse). T.A. Cargill [CAR 82] montre que cette stratégie ne peut entrainer d'étreinte fatale (deadlock) ni même de famine (La libération d'une fourchette doit se faire au bout d'un temps fini après son acquisition).

```
VAR
   MyNum : INTEGER ;
   Four1, Four2 : ModuleId ;
   Observateur : ModuleId ;

PROCEDURE Init () ;
VAR
   m : MESSINIT ;
BEGIN
   GetInit (m) ;
   MyNum := m.Num ;
   Observateur := m.Observe ;

   IF (ODD(MyNum))
      THEN
           Four1 := m.FourGauche ;
           Four2 := m.FourDroite ;
      ELSE
           Four1 := m.FourDroite ;
           Four2 := m.FourGauche ;
   END ;
END Init ;
```

Ce message d'initialisation est envoyé par le module Maitre d'hôtel.

- La phase d'initialisation du module Philosophe consiste d'abord à attendre le message d'initialisation du maître d'hôtel. Le comportement du philosophe consiste ensuite en une boucle

infinie d'étapes où il pense et mange. Pour manger, il s'adresse à chacune de ses fourchettes (*SendPrendre*). Il attend une réponse pour chacune de ses fourchettes. (service *ATTFOUR*).

```
SETSERV ({INIT}) ;
LesServices [WAIT ()] ;

LOOP
    JePense () ;

    cr := DUPLICATELINK (Myself,mprendre.Demandeur) ;
    mprendre.Quai := ATTFOUR ;
    SendPrendre (Four1,mprendre) ;

    SETSERV ({ATTFOUR}) ;
    LesServices [WAIT ()] ;

    UneFourchette () ;

    cr := DUPLICATELINK (Myself,mprendre.Demandeur) ;
    mprendre.Quai := ATTFOUR ;
    SendPrendre (Four2,mprendre) ;

    SETSERV ({ATTFOUR}) ;
    LesServices [WAIT ()] ;

    JeMange () ;

    SendPoser (Four1,mposer) ;
    SendPoser (Four2,mposer) ;
END ;
```

- La procédure *JePense* consiste à envoyer un message à l'observateur pour lui indiquer que le philosophe pense.

```
PROCEDURE JePense () ;
VAR
   m : MESSAFFICHER ;
BEGIN
    m.Etat := Penser ;
    m.Num := MyNum ;
    SendAfficher (Observateur,m) ;
END JePense ;
```

- Le *Philosophe* indique à l'Observateur qu'il est en train de manger. Il lui envoie donc un message (procédure *JeMange*).

```
PROCEDURE JeMange () ;
VAR
   m : MESSAFFICHER ;
BEGIN
    m.Etat := Manger ;
```

```
    m.Num := MyNum ;
    SendAfficher (Observateur,m) ;
END JeMange ;
```

- Le philosophe indique à l'obervateur qu'il a obtenu sa première fourchette. Il lui envoie un message contenant son numéro ainsi que son état.

```
PROCEDURE UneFourchette () ;
VAR
   m : MESSAFFICHER ;
BEGIN
    m.Etat := Fourchette1 ;
    m.Num := MyNum ;
    SendAfficher (Observateur,m) ;
END UneFourchette ;
```

4.5.2.3 L'observateur

Ce module observe l'état des philosphes. Lorsque le philosophe change d'état, le philosophe envoie un message à l'observateur lui indiquant son nouvel état. Le module *Observateur* possède un service *AffichерEtat.*

- ```
  PROCEDURE AfficherEtat () ;
  VAR
     m : MESSAFFICHER ;
  BEGIN
     GetAfficher (m) ;
     WriteString ('Le philosophe ') ;
     WriteInt (m.Num,3) ;
     CASE m.Etat OF
        Penser      : WriteString (' pense ') |
        Fourchette1 : WriteString (' a obtenu la fourchette 1') |
        Manger      : WriteString (' mange ')
     END ;
     WriteLn ;
  END AfficherEtat ;
  ```

  Le service *AffichерEtat* affiche le contenu du message à l'écran.

- Le module Observateur attend des messages des philosophes. Il comprend donc une boucle infinie d'attente de messages.

```
SETSERV ({AFFICHERETAT}) ;

LOOP
   LesServices [WAIT ()] ;
END ;
```
```

4.5.2.4 Le maitre d'hôtel

L'application "Les philosophes et les spaghettis" démarre avec l'exécution de ce module. C'est lui qui détermine le nombre de philosophes *nb* qu'il invite. En collaboration avec le gestionnaire de modules, il effectue la création des modules *Philosophes*, *Fourchettes* ainsi que le module *Observateur*.

- Le *Maitre d'hôtel* exécute les instructions suivantes:

```
WriteString ('Entrez le nombre de philosophes') ;
WriteLn ;
ReadInt (Nb) ;

mn.Demandeur := Myself ;
mn.Quai := CREEROBS ;
mn.Proto := 'Obs' ;
SendNewModule (ServeurModule,mn) ;

mn.Demandeur := Myself ;
mn.Quai := CREERFOUR ;
mn.Proto := 'Four' ;
FOR i := 1 TO Nb
  DO
    SendNewModule (ServeurModule,mn) ;
  END ;

mn.Demandeur := Myself ;
mn.Quai := CREERPHI ;
mn.Proto := 'Philo' ;
FOR i := 1 TO Nb
  DO
    SendNewModule (ServeurModule,mn) ;
  END ;

NbReponse := 0 ;

WHILE NbReponse # ((2*Nb)+1)
  DO
     SETSERV ({CREEROBS,CREERPHI,CREERFOUR}) ;
     LesServices [WAIT ()] ;
     INC (NbReponse) ;
  END ;

LancerPhilosophes () ;

SETSERV ({}) ;
LesServices [WAIT ()] ;
```

Le module saisit d'abord le nombre de philosophes *nb*. On peut remarquer que le module Maître d'hôtel effectue un envoi multiple de messages de création à destination du module GM. Le module attend ensuite les réponses du module GM en validant les trois services *CreerObs*, *CreerFour* et *CreerPhi*.

La procédure *LancerPhilosophes* consiste à envoyer à chaque philosophe son numéro ainsi que des liens vers ses fourchettes droites et gauches. Après avoir initialisé les différents philosophes, le module *Maître d'hôtel* n'a plus lieu d'être et meurt.

- Le service *CreerPhi* est activé à la réception d'un message provenant du module GM. GM a crée un module *Philosophe*. Un lien vers le module crée est renvoyé au module *Maître d'hôtel*. Le maître d'hôtel stocke ce lien dans un tableau *LesPhilos*.

```
PROCEDURE CreerPhi ;
VAR
   m : MESSREPLYNEWMODULE ;
BEGIN
   INC (DernPhilo) ;
   GetReplyNew (m) ;
   LesPhilos [DernPhilo] := m.Lien ;
END CreerPhi ;
```

Le lien vers le philosophe crée est mémorisé dans un tableau *LesPhilos*.

- Le traitement est analogue au traitement au service précédent mais il concerne maintenant la création d'une fourchette. GM répond ainsi à une demande de création de module *Fourchette*. Le Maitre d'hôtel stocke dans *LesFourch* les fourchettes qu'il a créees. Ces liens seront ensuite dupliqués pour les modules *Philosophes*.

```
PROCEDURE CreerFour ;
VAR
   m : MESSREPLYNEWMODULE ;
BEGIN
    INC (DernFour) ;
    GetReplyNew (m) ;
    LesFourch [DernFour] := m.Lien ;
END CreerFour ;
```

- Ce service est activé lorsque le module GM répond à la demande de création du module *Observateur*. Ce lien est mémorisé et va ensuite être dupliqué aux modules *Philosophes*.

```
PROCEDURE CreerObs ;
VAR
    m : MESSREPLYNEWMODULE ;
BEGIN
    GetReplyNew (m) ;
    Observateur := m.Lien ;
END CreerObs ;
```

- La procédure LancerPhilosophe consiste à envoyer aux philosophes un messages contenant:
  - Son numéro (*Num*),
  - Un lien vers le module Observateur (*Observe*),

– Deux liens vers ses fourchettes droites et gauches.

```
PROCEDURE LancerPhilosophes () ;
VAR
   i : INTEGER ;
   m : MESSINIT ;
BEGIN
   FOR i := 1 TO Nb
     DO
       cr := DUPLICATELINK (LesFourch [i],m.FourGauche) ;
       cr := DUPLICATELINK (LesFourch [(i MOD Nb)+1],m.FourDroite) ;
       cr := DUPLICATELINK (Observateur,m.Observe) ;
       m.Num := i ;
       SendInit (LesPhilos [i],m) ;
     END ;
END LancerPhilosophes ;
```

4.5.3 Remarques

- **La terminaison**
  L'algorithme exécuté pour le philosophe tourne indéfiniment. On pourrait affecter à chaque philosophe une quantité de spaghettis ou bien alors un nombre de sessions possibles où il peut manger. Dès qu'il a atteint cette limite, le module *Philosophe* meurt. Dans ce cas, les modules *Fourchettes* et *Observateur* n'ont pas à être modifier pour s'arrêter. Ces modules meurent automatiquement par absence de références:

  – Chaque module *Fourchette* meurt après le dernier des deux philosophes qui peut l'utiliser.

  – Le module *Observateur* meurt après le dernier philosophe qui termine de manger.

- Les modules *Philosophe*, *Fourchettes* et *Observateur* s'exécutent sur des sites quelconques. Pour manger, un philosophe pourra utiliser des fourchettes qui se trouvent sur d'autres sites.

4.6 Conclusion

L'aspect positif de cette partie de l'étude est d'avoir eu la possibilité de développer des applications **réparties**. Nous avons donné en exemple dans ce chapitre quelques applications, que nous avons développées. D'autres programmes ont été réalisés mais ne figurent pas dans cette thèse:

- **Graphe**
  Une solution au plus court chemin dans un graphe a été proposée. L'idée est d'affecter à chaque noeud, un module OMPHALE. Un ensemble de messages innonde le graphe, du noeud de départ vers celui d'arrivée. Le module arrivée trie les messages qu'il reçoit pour sélectionner le plus court chemin.

- **Recherche bibliographique**
  Le but de ce programme est de simuler les fonctions d'interrogation et d'emprunt sur les ouvrages d'un ensemble de bibliothèques. Les ouvrages et les bibliothèques sont modélisés par des **modules** qui simulent le fonctionnement des objets rèels.

La méthodologie de programmation que nous avons utilisée nous semble particulièrement appropriée à l'architecture OMPHALE.

Cependant, cette étude a permis de mettre en avant un certain nombre de besoins:

- Un langage de programmation OMPHALE
  Un module OMPHALE s'écrit à partir de trois modules MODULA-2. Cela alourdit fortement le travail de développement. Il parait donc nécessaire de définir un langage de programmation OMPHALE. Des travaux sont actuellement en cours sur la définition et l'implantation d'un tel langage [PLA 89]. Un préprocesseur est en cours de développement.
- Un metteur au point
  La phase de développement d'un logiciel est facilitée avec des outils de mise au point. Les metteurs au point en contexte centralisé sont maintenant parfaitement maitrisés. Beaucoup de metteurs au point sont symboliques et utilisent les possibilités de multi-fenêtrage des écrans à haute définition. Ces outils sont plus délicats à mettre en oeuvre dans une contexte multi-processus et qui plus est réparti.
- Une couche CORTEX
  Des travaux mériteraient d'être entrepris sur les modules systèmes de la couche CORTEX.

Chapitre 5
Evaluation

5.1 Etat du prototype

L'architecture OMPHALE a été développée sur plusieurs configurations matérielles.

- **Version bi-processeur**
  C'est la première version du système OMPHALE qui ait été développée. Elle a permis d'établir le modèle d'exécution du module OMPHALE à partir d'un couple de tâches SPART TC et TN. La notion de processus (ou tâches) est un outil de structuration du logiciel. Le noyau NERF définit une tâche TN par module. La tâche TN accéde uniquement aux ressources du module. Nous évitons ainsi la construction d'un noyau monolithique manipulant la totalité des structures de données NERF.
  Cette structuration en terme de tâches permet aussi d'exploiter les possibilités de parallélisme de l'architecture. On peut ainsi avoir sur un module de traitement (processeur), l'exécution du noyau NERF (ensemble des tâches TN), et sur un autre module de traitement l'exécution du code des modules (code des tâches TC). La similitude est alors frappante avec l'architecture du site 0 [DEL85b]. Malheureusement, l'U.G.M. du SPS7 est implantée dans le domaine privé du module de traitement. Il se révéle donc impossible de manipuler l'U.G.M. à partir d'un autre module de traitement. Les opérations de changement de contexte peuvent être coûteuses.
  Une version stabilisée de l'interface de programmation du noyau NERF a été établie. Cette version du système OMPHALE était **centralisée** et **mono-site**.

- **Version bi-processeur + Station de transport**
  Pour cette seconde version du système OMPHALE, notre objectif était de développer la station de transport et ainsi d'obtenir un véritable système réparti. La station de transport a donc été réalisée sous système SPIX en utilisant les "sockets" TCP/IP. Au niveau du noyau NERF, très peu de modifications ont été apportées, excepté au niveau de l'agence Routage où les messages distants sont passés à la station de transport. Sur cette version, nous avons intégré dans un même noyau les trois agences spécifiques NERF que sont l'agence Messages, l'agence Liens et l'agence Routage.

- **Version tri-processeur + Station de transport**
  Cette version comprend deux modules de traitement SPART et un module de traitement sous SPIX (Station de transport). Le premier module de traitement SPART supporte l'exécution des tâches TN, et son noyau inclut donc l'agence Routage. Le deuxième module de traitement SPART supporte l'exécution des tâches TC (agence Messages et Liens). Cette version est proche de la version 1, mais on a sur un troisième module de traitement, le système SPIX avec la station de transport.
  Une seconde version tri-processeur a été développée. Elle comprend toujours le module de traitement sous SPIX (station de transport). Chaque module de traitement SPART supporte l'exécution des tâches TC et TN. (Les trois agences sont inclues dans le noyau des MT). On

peut alors dire que chaque module de traitement SPART supporte l'exécution d'un **site virtuel** OMPHALE.

Le passage d'une version mono-site à une version multi-site nécessita le développement de la station de transport. Au niveau du noyau NERF, la répartition intervient peu. Elle se localise aux points suivants:

- Les messages OMPHALE destinés à des modules distants, sont passés à la station de transport. (primitive *Attendre* de l'agence Routage).
- La duplication d'un lien vers un module distant provoque une incrémentation du nombre de références et donc un envoi de messages de duplication vers ce module. (primitive *DuplicateLink* de l'agence Liens)
- La destruction effective des liens de l'environnement se fait à la fin de la phase d'activité. Un message de destruction est émis vers le module OMPHALE pointé par le lien. (primitive *Attendre* de l'agence Routage).

On peut constater que le noyau NERF est relativement indépendant de la configuration matérielle du site et notamment du nombre de modules de traitement. Deux raisons à cette souplesse:

- **Architecture modulaire du SPS7**
  Un des objectifs du CNET [FIN 81] était la conception d'une architecture matérielle modulaire et souple. Ajouter un module de traitement à la configuration consiste à enficher une nouvelle carte sur le SMBUS. La BULL SPS7/50 se révéle être une machine très souple et adaptée à l'étude d'architecture de système. On peut cependant regretter les performances modestes de la machine. Il sont en grande partie due aux disques (capacité de 56 Mégas octets et temps d'accès importants). Les dernières SPS7 commercialisées (SPS7/75) sont maintenant dotées de disques de capacités plus importantes (160 Mégas octets) avec des temps d'accès rapides.
- **Agences SPART**
  Le noyau SPART exploite la souplesse de l'architecture matérielle du SPS7. Le noyau SPART définit un ensemble d'objets. Chaque objet SPART est implanté dans une agence. Les opérations de manipulation de l'objet sont réalisées dans l'agence. Sur chaque module de traitement, un noyau SPART est chargé. Nous avons donc crée trois noyaux doté de différentes agences:
  - Noyau SPART + agence Messages + agence Liens
    Ce noyau supporte uniquement l'exécution des tâches Calcul.
  - Noyau SPART + agence Routage
    Ce noyau supporte l'éxécution des tâches Noyau.
  - Noyau SPART + agence Messages + agence Liens + agence Routage
    C'est un noyau NERF complet qui définit un site virtuel OMPHALE.

5.1.1 Quelques chiffres...

Notre noyau NERF a été construit à partir du noyau SPART. Trois agences spécifiques NERF ont été développées. Les primitives des agences Messages et Liens sont exécutées par les tâches TC. Les primitives de l'agence Routage sont appelées par les tâches TN. Le tableau ci-après indique le volume en code (source C et objet) de ces agences:

	Lignes source C	Taille objet
Agence Messages	500	8381
Agence Liens	352	4462
Utilitaires	150	2459
	1002	15302
Agence Routage	629	13142
Utilitaires	191	3417
Tâche Relai (créer TC)	169	3512
Tâche GT (créer TN)	110	3264
	1099	23335

Le noyau NERF est finalement obtenu en ajoutant **2000** lignes de programme C au noyau SPART. Actuellement, nous n'avons pas d'informations précises sur la taille en lignes C du noyau SPART. Nous pouvons seulement dire que le noyau SPART fait 170 kilos octets. Nous n'avons rajouté que 38 Kilos octets au noyau SPART pour obtenir le noyau NERF.

5.1.2 Caractéristiques du prototype

Le prototype pr'sente dans cette thèse est différent du modèle proposé par Jean-Marc Geib. Il posséde des limitations et certains aspects n'ont pas été développés car ils n'apportaient que peu d'élements à notre étude. D'autres aspects ont été étudiés tels qu'un mécanisme de rammase-miettes (Garbage Collector) de modules.

5.1.2.1 Limitations

Taille de l'environnement

La taille de l'environnement d'un module OMPHALE est limitée. En effet, l'environnement est représenté à partir d'un tableau dont la taille est fixée à la compilation du noyau. Cela pose des problèmes pour la réalisation de modules **catalogue**. On peut définir un module catalogue comme un module

dont la fonction principale est de stocker des liens et d'en restituer une copie à la demande. L'exemple typique que nous avons étudié est le gestionnaire de noms symboliques de la couche CORTEX. On peut imaginer d'autres modules de ce type tels que par exemple un module répertoire d'une arborescence de fichiers.
Des solutions sont envisageables à cette limitation:

- Nous conservons le tableau tel qu'il est défini actuellement. Lorsque cet environnement est saturé, le noyau alloue un nouveau tableau qui est chaîné au précédent.
- L'environnement est implanté à l'aide d'une liste chaînée de liens. A chaque maillon de la liste correspond un lien. Les liens sont alors identifiés par le pointeur du maillon.

Réutilisation des noms

Le nom interne de module est construit à partir de l'identificateur de la boîte aux lettres SPART MBN. SPART réutilise les identificateurs de boîtes aux lettres. Les noms de modules ne sont donc pas uniques dans le temps: une valeur de nom peut repérer un module n'existant plus, ou un module différent. On peut lever cette contrainte en gérant au niveau de NERF une table de correspondance entre un identificateur unique et un identificateur de boîte aux lettres SPART.

5.1.2.2 Mécanismes non implantés

Priorités sur les services

Nous n'avons pas défini au niveau des files d'attente de la zone de réception des priorités sur les files. Lorsque plusieurs files d'attente sont ouvertes et qu'elles contiennent au moins un message, le noyau choisit une de ces files. En fait, le noyau choisit la file contenant le message le plus ancien (se trouvant dans la zone de réception depuis le plus longtemps. Le module OMPHALE ne doit pas tenir compte de cet algorithme de choix qui est fait dans le noyau.

Mécanismes d'attente limitée

L'attente limitée se révéle primordiale dans la problématique de la tolérance aux pannes. Un module A envoie un message à un module B et attend de B une réponse. Si cette réponse n'arrive au bout d'un certain temps, le module A peut considérer que B est en panne et associer un traitement particulier dans ce cas. Il est donc nécéssaire de pouvoir se mettre en attente un temps fixé. Si ce temps expire, le noyau renvoie alors au module un code d'erreur.
Les mécanismes d'attente limitée sont aussi utiles dans les applications temps-réel. SPART est un noyau temps-réel. Il ne nous parait pas insurmontable d'implanter un tel mécanisme au niveau du noyau NERF en utilisant les fonctions temps-réel de SPART. L'unité de temps SPART est la milliseconde.

5.1.2.3 Extensions

Ramasse-miettes

Nos expériences ont montré que le style de programmation induit par l'architecture OMPHALE conduisait à un nombre important de modules. La solution que nous proposons est coûteuse en tâches: Deux tâches SPART par module. Il est donc indispensable de ne garder en mémoire que les modules réellement actifs. Nous avons proposé un mécanisme de destruction des modules en attente et qui ne sont plus référençables. Ce mécanisme est basé sur le comptage par la tâche TN du nombre de références (liens se trouvant en dehors du module). Cette technique est trop primitive et ne permet pas de détecter les boucles de références entre modules. De plus, cette solution se révele inadaptée dans le cadre d'une étude sur la tolérance aux pannes.
L'étude d'un ramasse-miettes (Garbage Collector) réparti devrait être mené.

Nommage de liens

Dans cette thèse, nous avons proposé un mécanisme de noms unique de liens dans l'environnement d'un module. Le module OMPHALE manipule les liens de son environnement au moyens de noms locaux. Au fil du temps, un nom local doit désigner un module et un seul.

5.1.3 Ce qui a bien fonctionné...

Les aspects positifs sont nombreux! En particulier, nous avons pu développer des applicattions **réparties** au dessus de l'architecture OMPHALE. Certaines de ces applications ont été présentées dans cette thèse:

- Deux implantations de la structure de données *Pile*,
- Une solution répartie au problème des philosophes et des spaghettis,
- Une ébauche de base de données documentaire, (non présentée dans cette thèse).
  Chaque livre est matérialisé par un module qui renferme un titre, un nom d'auteur et l'état du livre (en bibliothèque, emprunté ou perdu). Des opérations d'interrogations sur l'auteur ou le titre sont disponibles.
  Des modules bibliothèques qui sont en fait des catalogues de modules livres. Une bibliothèque posséde un lien vers l'ensemble des livres qu'elle détient. La limitation de la taille de l'environnement pose alors un problème au niveau des modules bibliothèques.
  L'arrêt du système ne doit pas entraîner la destruction de modules. Le stockage des modules en mémoire permanente n'a pas étudié. Cette application a permis aux modules lecteurs d'interroger les bibliothèques. Des questions sont élaborées et un arbre d'évaluation est construit.
- Une solution répartie et parallèle à la recherche du plus court chemin dans un graphe, (non présentée dans cette thèse) Après avoir réalisé les structures de donnée Pile et Arbre (base de données documentaires), nous avons étudié la représentation du graphe. Les arcs du graphe

sont représentés à partir des liens et les noeuds du graphe à partir de modules OMPHALE. Cet exemple a permis de mettre en œuvre un algorithme avec diffusion de messages.

Une version minimale de la couche CORTEX a été développée: Elle comprend:

- Un serveur de noms,
- Un serveur de modules,
- Une station de transport,
- Un interpréteur de commandes.

Ces modules fonctionnent suivant le modèle Client-Serveur. Ce modèle a eu pour conséquence l'utilisation fréquente de la technique du long-retour [STR 81].
Une étude plus complète serait à mener sur d'autres modules de cette couche pour obtenir ainsi un système d'exploitation complet.

Le noyau répond presque complétement aux spécifications de l'architecture. Les caractéristiques suivantes ont été implantées et validées:

- La communication **asynchrone** par messages de taille variable,
- Le schéma d'exécution du module OMPHALE,
- L'attente sélective de messages,
- Le nommage et la protection par un mécanisme de liens,
- l'édition de liens **dynamique** par transfert de liens dans les messages.

Ce travail a été mené conjointement à la spécification de l'architecture par Jean-Marc Geib. Une interface du noyau NERF a été définie. D'autre part, l'implantation a permis de préciser les points suivants de l'architecture OMPHALE:

- **Destruction de modules**
  L'étude a montré différent mécanismes de destruction de modules. La destruction de module peut être explicite et demandée par le module lui même (suicide) ou bien par un autre module (Envoi d'un message sur le service Destruction du module). Elle est implicite lorsque le noyau s'aperçoit que le module ne peut redevenir actif (absence de références).
- **Transfert de liens dans les messages**
  La structure interne du message OMPHALE est scindée en deux zone: Une zone utilisateur qui est une suite d'octets qui n'est pas interprétée par le noyau, et une zone Liens qui contient la liste des liens du messages. Le noyau NERF définit des primitives d'ajout et suppression de liens de cette zone.
- **Reprise après un *Wait***
  La programmation d'application montre rapidement que l'exécution d'un service est réalisée

en collaboration avec d'autres modules. Il est donc nécessaire de pouvoir exécuter la primitive *Wait* pendant un service. Ceci permet de restreindre le nombre de services d'un module.

5.2 Structure du noyau

Le développement de logiciels importants (plusieurs milliers de lignes de source) pose des problèmes de conception. Un système d'exploitation est un logiciel important. Les concepteurs de systèmes d'exploitation ([DIJ68a], [HOL 83], [COM 83] et [TAN 87]) ont été particulièrement sensibilisés par ce problème. Plusieurs points sont à étudier:

- **Parallélisme de développement**
  Le développement de compilateurs, système et d'autres logiciels importants ne peut être l'oeuvre d'une seule personne. Il est impératif de recourir à un découpage du logiciel. Ce découpage permet une structuration du logiciel et facilite donc sa conception. La réalisation des différents composants (modules) est confiée à plusieurs programmeurs. Ce découpage doit permettre un parallélisme de développement entre les différents programmeurs.

- **Maintenance**
  La phase de maintenance est très importante dans le cycle de vie d'un logiciel. Il existe deux raisons principales pour maintenir un logiciel:
  i) Réparer les erreurs (buggs) détectées à l'utilisation du logiciel et qui seraient passées au travers du jeu de test du logiciel (avant son exploitation).
  ii) Ajouter de nouvelles fonctionnalités au logiciel. De nouveaux besoins apparaissent fréquemment après la mise en service du produit.
  La maintenance du logiciel est souvent effectuée par d'autres personnes que celles qui ont participé au développement.

- **Portabilité**
  Les logiciels sont souvent utilisés sur plusieurs systèmes ou architectures. Il est souhaitable que le transport du logiciel sur un nouveau système ne demande pas une réécriture compléte du logiciel. Les éléments dépendants de l'architecture sont isolés dans des modules bien spécifiques. Le transport du logiciel ne nécéssite, alors, que la réécriture de ces modules.

Les langages de programmation proposent maintenant des outils de "découpe" du logiciel. Ces outils sont les modules ou paquetages (ADA [Dod 83], MODULA-2 [WIR 83]). On regroupe dans un même module une collection d'objets qui sont conceptuellement liés. Les modules comprennent deux parties:
i) Une *spécification* où l'on définit les éléments visibles du module.
ii) Une *implantation* où l'on décrit la représentation des objets du module.

L'approche hiérarchisée consiste à définir le logiciel en un certain nombre de niveaux. Chaque niveau a des fonctions bien précises à réaliser. A chaque niveau, on fait correspondre des modules. La définition de ces modules précise l'interface du niveau. L'implantation des modules décrit la réalisation des primitives du niveau. L'implantation fait appel aux modules des couches inférieures.
E.W. Dijkstra [DIJ 68] a été un des premiers à utiliser cette approche pour développer le système **THE**

(Technische Hogeschool Eindhoven). Cette méthodologie a ensuite été souvent réutilisée pour le développement de systèmes plus récents.

Les systèmes d'exploitation répartis [GUI 84] [RAS 84] sont maintenant structurés sur le modèle suivant:

- **Applications**
- **Serveurs du système**
- **Noyau minimal**

OMPHALE reprend cette structuration:

- Noyau NERF
  Le noyau minimal NERF assure l'enchaînement et la communication des modules du site. La protection est garantie par le mécanisme de nommage : Le Lien.
- Couche CORTEX
  Les fonctions du système sont réalisées par les modules de la couche CORTEX. La gestion des fichiers, gestion mémoire est assurée par les modules du CORTEX. Cette couche est répartie sur les sites de la configuration.
- Applications
  L'application est découpée en module s'exécutant sur les différents sites. Les modules communiquent et se synchronisent par l'envoi asynchrone de messages.

5.2.1 Le noyau monolithique XINU

XINU [COM 83] est un système multi-programmé qui a été développé sur LSI/11. Ce système a été essentiellement développé à des fins pédagogiques pour illustrer les fonctions d'un système d'exploitation. XINU définit les couches suivantes:

1. **Matériel**
   XINU a été initiallement développé sur un LSI 11/2 (Digital Equipement Corporation). Le système XINU a également été porté sur le processeur MC 68000 (Motorola). L'architecture matérielle est centralisée et comprend un processeur, de la mémoire et des périphériques.
2. **gestion mémoire**
   Il s'agit dans cette couche de gérer la mémoire physique. Cette couche propose des fonctions d'allocation et libération mémoire. Le système XINU ne gère pas de mécanisme de "swapping" ni de mémoire virtuelle.
3. **processus**
   Ce niveau détaille le concept de processus. Le partage du processeur entre les différents processus est assuré à ce niveau. Les fonctions internes du noyau (choix d'un nouveau processus à exécuter, changement de contexte) sont décrites dans cette couche. Pour réaliser ces fonctions, le noyau décrit les informations associées à chaque processus dans la table des

processus.
On trouve également à ce niveau les primitives de création et destruction de processus.

4. **communication inter-processus**
   Cette couche implante les outils de synchronisation et communication standard du système XINU. La synchronisation se fait par sémaphore ou par envoi de messages.

5. **gestion horloge temps-réel**
   Une horloge est un mécanisme matériel qui émet des impulsions à intervalles réguliers avec une grande précision. Ce dispositif permet d'implanter des horloges logiques . Il sert en particulier au noyau pour limiter le temps d'exécution de chaque processus et ainsi de pouvoir effectuer la réquisition de l'unité centrale (réquisition).

6. **pilote d'entrées-sorties**
   Cette couche gère les accès aux périphériques. Elle a trois fonctions principales:

   - Cacher les détails de la programmation souvent obscure des périphériques.
   - Partager les périphériques entre les différents processus du système.
   - Fournir une interface unique et flexible d'utilisation des périphériques.

7. **système de fichiers**
   Les fichiers sont des objets permanents qui ont une durée de vie plus longue que celle des processus. Les fichiers sont stockés sur des supports secondaires tels que des disques ou disquettes. Le système de fichiers gère les fichiers en proposant des primitives d'ouverture (open) de lecture/écriture (read, write) et fermeture (close). Le système de fichiers assure également des fonctions de protection en vérifiant que les processus ont un accès correct sur les fichiers.

8. **processus utilisateurs**
   Ce niveau comprend l'ensemble des processus et programmes crées par les utilisateurs.

Le découpage en terme de couches proposé par XINU est extrêmement fin. Toutes les fonctions du système sont bien isolées dans les différentes couches. Cependant, cette décomposition est **statique**, c'est à dire qu'elle n'a d'existence que durant la phase de conception et de développement du système. La phase de développement aboutit à un noyau monolithique et rigide où toutes les fonctions sont intégrées dans un binaire exécutable. La structure hiérarchique a **disparu**. MINIX propose une structuration qui demeure à l'exécution.

5.2.2 Le noyau structuré de MINIX

MINIX a été développé par A.S. Tanenbaum [TAN 87]. MINIX est un système d'exploitation UNIX [RIT 74] version 7. L'objectif de Tanenbaum est de diffuser largement les concepts du système UNIX et les fonctions d'un système d'exploitation. Pour ce faire, MINIX a été initiallement développé sur IBM-PC et des disquettes accompagnent le livre de Tanenbaum.

MINIX [TAN 87] est structuré en cinq couches qui sont:

- **matériel**
  Le matériel se compose d'un micro-ordinateur IBM-PC compatible. Il est bâti autour d'un micro-processeur INTEL 8088 auquel sont connectés mémoire, clavier, écran, disque et autres périphériques. La connection du micro-processeurs avec les autres composants se fait par un **bus système.**
  Actuellement, ce micro-ordinateur connait un grand succès. Les logiciels ou systèmes fonctionnant sur cet ordinateur s'adressent donc à un vaste public.

- **gestion de processus**
  Cette couche définit le modèle de processus séquentiel. Les processus MINIX communiquent et se synchronisent avec un mécanisme d'envoi synchrone de messages (rendez-vous [DoD 83]). De plus cette couche convertit les interruptions matérielles en messages destinés aux tâches d'entrées-sorties.

- **entrées-sorties**
  Les échanges avec les périphériques sont gérés à ce niveau. A chaque type de périphérique, correspond un processus qui pilote ce périphérique. Pour les distinguer des autres processus du système, MINIX utilise le terme de tâche (task) pour désigner les processus d'entrées-sorties. Ces tâches sont des processus **serveurs** de périphériques. Ces serveurs sont accessibles par les processus serveurs de la couche suivante

- **processus serveurs**
  Ce niveau comprend deux processus **serveurs** qui fournissent des services du système aux modules **clients.**
  i) Le **gestionnaire mémoire** (memory manager MM) traite tous les appels au système qui nécessitent la gestion mémoire (*fork, exec*).
  ii) Le **serveur de fichiers** (file system FS) exécute toutes les fonctions systèmes d'accès aux fichiers (*open, read, write, close...*).

- **application**
  Cette couche comprends les processus utilisateurs tels que l'interpréteur de commandes, les éditeurs, compilateurs et les programmes utilisateurs.

La structuration hiérachique de MINIX a facilité la conception même du système. Cette hiérarchie demeure à l'exécution du système, c'est à dire que contrairement à XINU les couches continuent à demeurer à l'exécution du système. Une des raisons est que le concept de processus est utilisé pour le développement du système. (tâches d'entrées-sorties et processus serveurs)

Le noyau MINIX (code exécutable) contient la couche de gestion des processus et le code des tâches d'entrées-sorties

5.2.3 La structuration du noyau NERF

Le système OMPHALE est structuré en quatre couches:

1. **Matériel**

Le matériel se compose d'un ensemble de sites (SPS7 BULL) connectés à un réseau local (ETHERNET).

2. **Noyau**
   Le noyau NERF assure l'enchainement et la communication entre les modules du sites. Nous avons retenu la solution d'écrire une couche au dessus du noyau SPART de la SPS7. Un des inconvénients de cette technique est la non-maitrise des mécanismes d'exécution du système SPART. Les mesures de performances n'ont pu être faites.

3. **CORTEX**
   Cette couche comprend un ensemble de modules systèmes qui fournissent les fonctions évoluées des systèmes d'exploitation. L'objectif est de ne laisser dans le noyau qu'un minimum de primitives.

4. **Application**
   Une application OMPHALE est structurée en module s'exécutant sur les différents sites de la configuration. La décomposition modulaire est un outil de conception de l'application mais aussi une structuration de l'exécution de l'application.

La structuration proposée par OMPHALE est proche de celle proposée par MINIX, système centralisé. Les systèmes répartis tels que CHORUS [GUI 84], ACCENT [RAS 84] sont structurés de la même manière.

5.2.4 Modèle Client-Serveur

Une des tendances actuelles dans la conception des systèmes (répartis en particulier) est de développer un noyau **minimal**. Un certain nombre de services ne sont plus rendus par le noyau, mais par des processus systèmes qu'on appelle des processus **Serveurs**. Pour faire exécuter un service, le processus **Client** doit adresser une requête au processus **Serveur**. La requête au processus **Serveur** peut se réaliser par envoi de messages. MINIX propose deux processus **Serveurs**:

- *Memory Manager*
  Les appels systèmes MINIX nécéssitant des opérations d'allocation ou libération mémoire sont réalisés par ce serveur.
- *File System*
  Les appels système ayant trait aux fichiers sont réalisés par le processus *File System*.

La communication avec les processus **Serveurs** MINIX se fait par l'envoi **synchrone** de message qui est l'outil de communication inter-processus du système MINIX.

Le modèle Client-Serveur s'intégre parfaitement aux systèmes répartis. Les concepteurs de système répartis ont donc suivi la tendance en confiant à des processus **Serveurs**, la majorité des fonctions systèmes.

5.2.4.1 OMPHALE

Réalisation du module OMPHALE

NERF est développé au dessus du noyau SPART. Les objets de base du noyau NERF sont donc construits à partir des objets SPART. Le noyau NERF définit un objet unique qu'on appelle module. Le module se décompose en deux tâches SPART s'exécutant en parallèle:

- La tâche calcul TC exécute le code des services du modules et gère la stratégie d'exécution des services. (Code utilisateur).
- La tâche noyau TN assure l'émission des messages OMPHALE du module OMPHALE. (fonction de routage). TN place dans la zone de réception, les messages reçus par le module. De plus, la tâche TN contrôle les références existantes sur le module.

Les tâches TC et TN peuvent s'exécuter sur deux modules de traitements différents. Cette exécution est similaire à celle développée sur le site 0 [DEL85b]. Le site 0 est contruit autour de deux processeurs:

- Un processeur Noyau (PN).
  Le changement de contexte rapide des processus du processeur de calcul PC est supporté par PN. Le code du noyau NERF est exécuté sur le processeur noyau PN.
- Un processeur Calcul (PC)
  Exécution du code utilisateur des modules OMPHALE.

Structuration en agences

L'agence est un outil de structuration proposé par le rapport SCEPTRE [SCE 82]. Chaque agence définit un type d'objet et l'ensemble des primitives se rattachant à l'objet. Pour implanter ces primitives, l'agence peut utiliser les primitives d'une autre agence. Le noyau peut être étendu de deux façons:

- **Ajout de nouvelles primitives sur un objet**
  Dans ce cas, il faut intégrer à une agence existante les nouvelles primitives.
- **Création d'un nouvel objet**
  Il faut créer une nouvelle agence et définir l'ensemble des primitives agissant sur l'objet.

Pour valider l'extension, un nouveau noyau est obtenu par édition de liens incluant le code de l'agence.

Les fonctionnalités NERF sont implantées dans trois agences:

- **Agences Messages**
  Le message OMPHALE est défini par cette agence. Les messages OMPHALE sont stockés dans la zone d'émission du module. Les primitives de l'agence Messages ont trait à la gestion de la zone d'émission. Le module peut ainsi placer, retirer des messages de sa zone d'émission. Les primitives de l'agence Messages sont exécutées par la tâche TC du module OMPHALE.

- **Agences Liens**
  La structure de donnée Lien est déclarée par l'agence Liens. Les liens du module sont rangés dans l'environnement du module OMPHALE. L'agence définit un ensemble de primitives agissant sur l'environnement du module. Les primitives de l'agence Liens sont exécutées par la tâche TC.

- **Agences Routage**
  La zone de réception du module est définie par l'agence Routage. L'ajout de messages à cette structure, la destruction de messages dans la zone de réception sont réalisés par les primitives de l'agence Routage. La tâche TN exécute les primitive de l'agence Routage.

Le mécanisme d'**abstraction** [BIS 86] est un concept de programmation. Les langages de programmation proposent des outils facilitant le principe d'abstraction. Nous citerons les paquetages ADA [DoD 83], les modules MODULA-2 [WIR 83]. Les types de données peuvent être déclarés dans ces unités et cachés (**private** en ADA, types cachés dans l'implémentation du module en MODULA-2).
Le mécanisme d'agence SCEPTRE permet l'implantation de types de données abstrait au niveau du noyau d'un système. La représentation des objets est cachée dans l'agence. Les objets OMPHALE (Message, environnement, zone d'émission et réception) sont cachés dans les agences.
L'exécution des primitives de l'agence se fait en mode système. Les données peuvent être encapsulées au niveau de l'agence et ainsi être inaccessibles en mode utilisateur. Dans le noyau NERF, le lien est l'entité à protéger. L'environnement du module, les liens ne sont manipulables qu'au moyen des primitives de l'agence Liens. Les primitives d'une agence sont supposées **fiables.**

Correspondance entre objets SPART et OMPHALE

Notre noyau NERF est construit à partir d'une couche développée au dessus du noyau SPART. Cette couche et le noyau SPART sont intégrées dans un même et unique noyau. Les objets de NERF (module, message) sont définis à partir d'objets SPART:

- Un module OMPHALE est réalisé à partir d'un couple de tâche SPART, de deux boîtes aux lettres SPART et un segment pour les zones d'émission et réception. Cela permet d'introduire un parallélisme au niveau de l'exécution du module OMPHALE. L'inconvénient majeur de cette solution est la grande consommation de ressources. Un module OMPHALE coûte deux tâches à l'exécution. Lorsque le nombre de modules OMPHALE augmente, le nombre de tâches SPART augemente deux fois plus vite!

- Le message OMPHALE se décrit par un message SPART contenant un pointeur sur un emplacement mémoire dans un segment SPART. Dans la section **Performances**, nous verrons qu'un transfert de message OMPHALE se matérialise par plusieurs envois de messages SPART.

Le constructeur ne nous a jamais fourni de documentation sur l'implantation du noyau SPART. De plus, nous n'avons aucune informations quantitatives sur le noyau SPART en terme de temps d'exécution ou bien alors en ressources mémoires utilisées.

5.2.5 Evaluation de la structuration de NERF

5.2.5.1 Avantages

- **Parallèlisne entre TC et TN**
  Au cours de l'exécution d'un service, la tâche TC est active. Au même moment, des messages OMPHALE peuvent arriver au module. L'arrivée d'un message peut être simultanée à un traitement.
- **Portabilité au dessus du noyau SPART**
  La portabilité du noyau NERF n'est assurée qu'au dessus du noyau SPART et à la rigueur de systèmes dérivés du rapport SCEPTRE.
- **Souplesse de la configuration**
  L'architecture modulaire de la SPS7 permet de modifier rapidement la configuration. La structuration de NERF s'adapte particulièrement à la souplesse de l'architecture de la SPS7.
- **Agence**
  - Modularité
    Le concept d'agence oblige le concepteur du système à structurer l'interface de son système. L'agence a permis de bien classifier les primitives et de les affecter à des agences.
  - Protection
    La notion d'agence apporte la protection nécéssaire pour l'exécution du noyau et ainsi protéger des structures telles que les liens, l'environnement ou bien la zone de réception.

5.2.5.2 Inconvénients

- **Changement de contexte**
  La décomposition du module OMPHALE en deux tâches SPART provoque des changements de contexte. (voir le paragraphe Changement de contexte). Ce qui montre l'intérêt d'architectures spécialisées à changement de contexte rapides telles que le site 0 [DEL85b].
- **Complexité de NERF masquée**
  NERF utilise un nombre limité de primitives SPART. Le noyau SPART fait à lui seul près de 170 Koctets. Nous n'utilisons pas la globalité des fonctions de ce noyau.
- **Migration fonctions SPART vers OMPHALE**
  Un certain nombres de fonctionnalités sont assurées par le noyau SPART. La création d'une tâche (chargement du code, allocation mémoire) sont des primitives internes du noyau SPART. Ces fonctions sont à remplir par la couche CORTEX de OMPHALE. La solution serait de ré-écrire le code de certaines agences dans des modules de la couches CORTEX.

5.3 Performances

5.3.1 Transfert de messages

Le transfert de messages locaux est réalisé complétement par le noyau NERF. Les messages distants sont transférés avec le concours du module système: *La station de transport.* CHORUS ([GUI 84]) définit également un acteur appelé *acteur station de transport* A.S.T.
Il est quelque fois reproché aux systèmes à communication par messages l'inefficacité. Dans cette section, nous tenterons d'évaluer le coût d'un envoi de message OMPHALE. Dans un premier temps, nous évaluerons le cout du transfert de messages OMPHALE en nombre d'envois de messages SPART.

Un message SPART est émis de TC vers TN à la fin de la phase d'activité. Au pire, il y a donc un message SPART pour un message OMPHALE. Les cas des messages locaux ou distants sont à étudier.

- **message local**

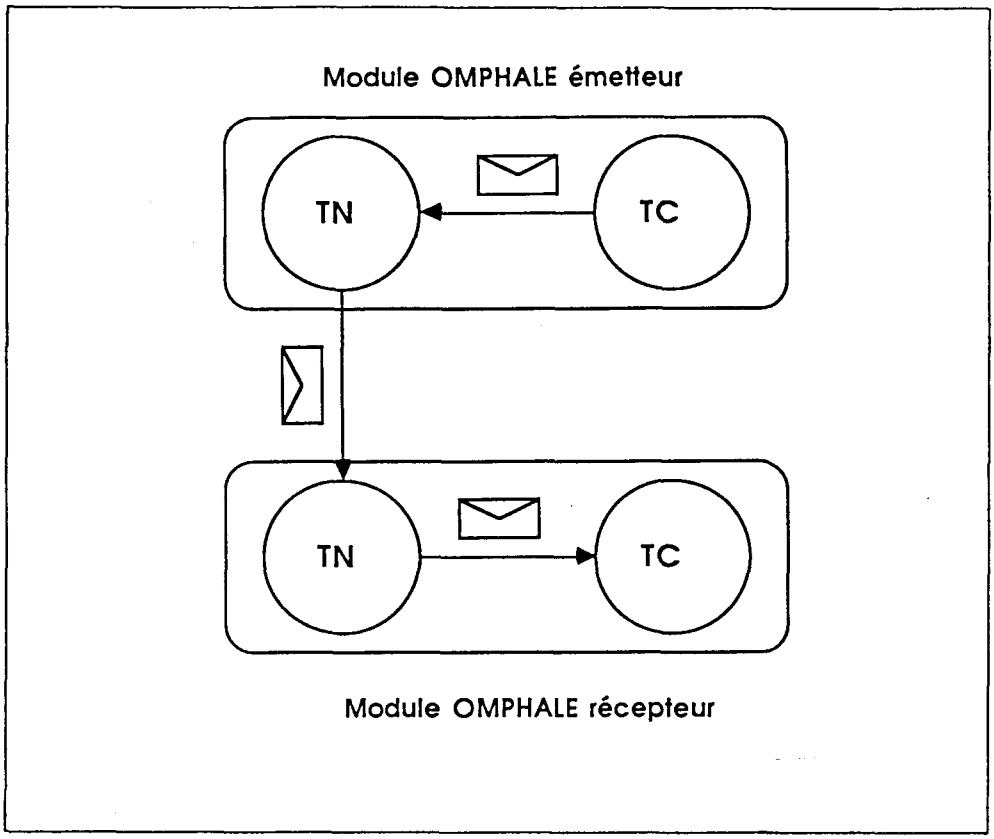


Figure 13. Transfert de message local

La tâche TN de l'émetteur envoie un message SPART (pointeur sur le message OMPHALE) vers la tâche TN du module récepteur. La tâche TN du récepteur envoie un message SPART vers TC au moment du réveil. Pour un message OMPHALE local, nous avons donc trois envois de messages SPART.

- **message distant**

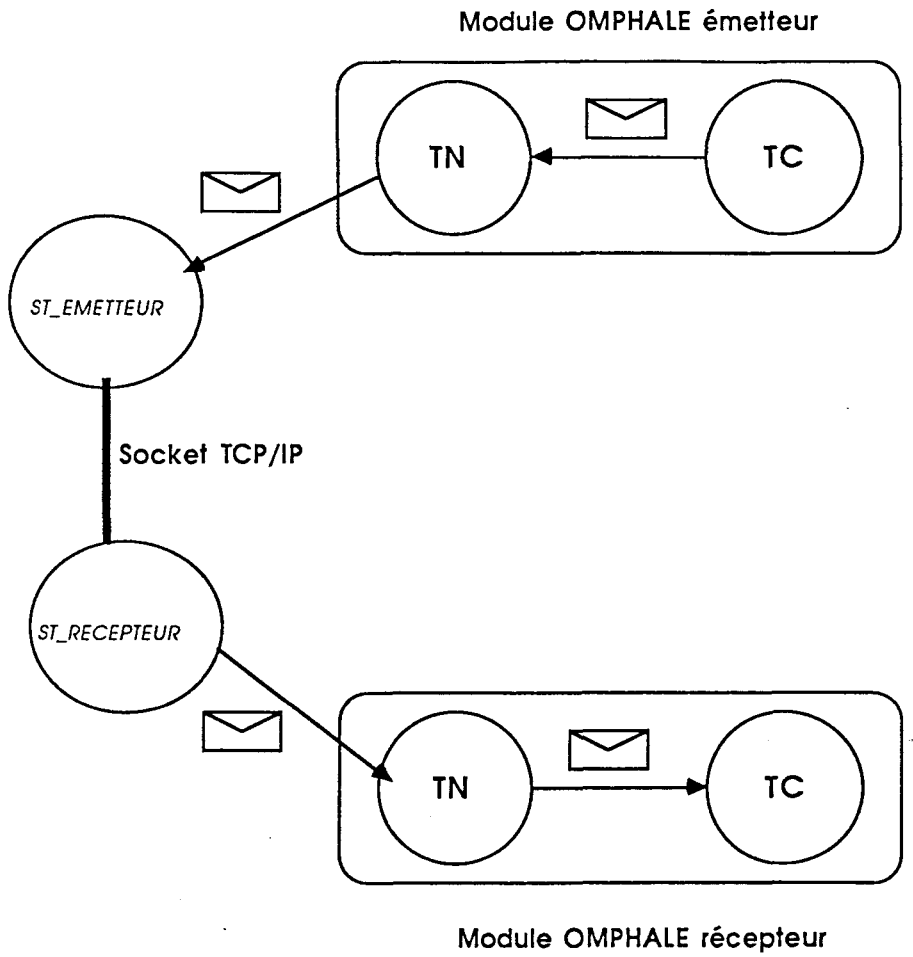


Figure 14. Transfert de message distant

La tâche TN de l'émetteur envoie un message SPART vers la station de transport (processus SPIX *ST_EMMETTEUR*). Le contenu du message SPART est émis sur la socket TCP/IP. Le processus SPIX *ST_RECEPTEUR* reçoit le message sur la socket. Il envoie un message SPART vers la tâche TN du module récepteur. Dans ce cas, nous avons quatre envois de messages SPART plus un envoi sur la socket TCP/IP.

L'évaluation du coût de transfert peut se faire à partir du nombre de copies du message OMPHALE. NERF essaie de minimiser le nombre de copies du message OMPHALE. Le principe est de véhiculer un pointeur vers le message. Deux cas sont encore à étudier:

- **message vers un module local**
  A l'émission du message, le message est copié de l'espace d'adressage du module émetteur dans le segment système. Un pointeur vers ce message est mémorisé dans la zone d'émission. Ce pointeur est envoyé dans la zone de réception du module destinataire. Le message est recopié dans l'espace d'adressage du module destinataire. Pendant le transfert, il n'y a donc pas eu de recopie du message.

- **message vers un module distant**
  Le pointeur vers le message OMPHALE est passé à la station de transport. La station de transport du site émetteur **recopie** le contenu du message sur la socket TCP/IP. La station de transport du site récepteur **recopie** le contenu de la socket dans un message OMPHALE. C'est ensuite le pointeur vers ce message qui est émis vers la zone de réception du module récepteur. Pour un message distant, il y a donc **deux** copies supplémentaires du message.

5.3.2 Changement de contexte

Dans cette section, nous évaluons le nombre de changements de contexte nécessaire pour l'exécution d'un service. L'arrivée d'un message OMPHALE provoque:

- L'arrivée d'un message SPART sur la tâche TN d'où le réveil de la tâche TN (changement de contexte sur TN). TN place le message dans la zone de réception et analyse les conditions de réveil du module. Si le module peut être réveillé, TN émet un message vers la tâche TC.
- L'arrivée du message de réveil sur TC provoque un changement de contexte sur la tâche TC. La tâche TC traite le message OMPHALE reçu.

L'arrivé d'un message OMPHALE au module provoque donc deux changements de contexte. L'étude d'architecture à changement de contexte rapide prend alors toute sa valeur. E. Delattre [DEL85b] a développé le site 0 d'OMPHALE où le changement de contexte de module est **instantané**. Le site 0 est équipé de deux processeurs: processeurs:
i) Le processeur Calcul où s'exécute le code des modules OMPHALE.
ii) Le processeur Noyau
La préparation du prochain module à exécuter, le chargement des registres MMU d'accès à la mémoire sont à la charge du processeur Noyau. Le changement de contexte sur le processeur Calcul est quasi-instantané:

5.3.3 Mesures

5.3.3.1 Ressources mémoire

Il est maitenant intéressant d'évaluer précisément les ressources affectées pour chaque module OMPHALE. Mais préalablement, nous devons quantifier les ressources affectées à chaque objet SPART. Pour chaque objet, SPART alloue un descripteur de 32 octets indiquant le type et les valeurs de l'objet.

- **Boîte aux lettres**
  Pour une boîte aux lettres, SPART réserve une zone de stockage pour les messages reçus. Cette zone est allouée dans le segment système (respectivement local et global pour les boîtes aux lettres locales et globales). La taille de cette zone est égale au nombre de messages (arrondie à la puissance de 2 immédiatement supérieure) que multiplie 8 octets, le tout arrondi à un nombre entiers de granules (granules de 32 octets pour le segment système local et des granules de 64 octets pour le segment système global).

- **Tâches**
  Une tâche SPART est crée à partir d'un programme (objet SPART). Un programme SPART comprend:

  – Un segment de code,

  – Un segment de donnée.

  En plus du programme, la tâche SPART utilise les ressources suivantes:

  – Un segment de pile,

  – Un segment des données de la tâche,

  – Une zone de sauvegarde des registres de l'UGM,

  – Un contexte de 128 octets pour la tâche SPART.

  Pour une tâche SPART, nous avons donc 6 objets SPART, y compris l'objet programme.

- **Segment mémoire**
  SPART associe un descripteur d'objet pour un segment mémoire de 256Koctets. Pour les segment de taille supérieure, SPART ajoute un descripteur d'objet par tranche de 256 Koctets.

Pour un module OMPHALE, nous allouons donc:

- Deux boîte aux lettres SPART MBN et MBC
- La tâche TC

  – Un objet programme pour le code utilisateur (3 objets SPART)

  – L'objet tâche (3 objets SPART)

- La tâche TN
  Le code de la tâche TN est réentrant. Nous ne comptabiliserons donc que les objets de l'objet tâche, ce qui donne 3 objets SPART.

Ce qui donne en tout 11 objets SPART (11 * 32 = 352 octets) pour un module OMPHALE.

- **La boîte aux lettres MBC**
  La boîte aux lettres SPART contient au plus deux messages SPART. La zone de stockage nécessaire est donc de 16 octets arrondie à 64 octets (1 granule).

- **La boîte aux lettres MBN**
  Pour la boîte aux lettres MBN, on a 256 octets (4 granules) de zone de stockage des messages (128 * 2 = 256 octets)
- **La tâche TC**
  - Un objet programme qui comprend le code du module (environ 30 Koctets) et un segment de données initialisées (2 Koctets). Les mesures sont fonction de la complexité du module.
  - Segment de données plus le tas (environ 10 Koctets)
  - Segment de pile (2 Koctets)
  - Contexte d'une tâche SPART (128 octets)
  - Zone de sauvegarde des registres de l'UGM (400 octets)
  - Zone de communication TC-TN de 83 octets.
  - Environnement du module de 32 liens (32 * 11 = 352octets)

  Ce qui donne pour la tâche TC environ 45 Kilos octets.
- **La tâche TN**
  - L'objet programme est partagé avec l'ensemble des autres TN. Nous ne comptabiliserons donc pas de mémoire.
  - Pile de 2 Kilos octets.
  - Segment de données et Tas (10 Koctets).

Pour un module, environ 60 Kilos octets sont alloués.

5.3.3.2 Temps

L'agence horlogerie SPART définit un certain nombre d'outils de mesure. L'unité de mesure est la **milliseconde**. Cette unité se révelera trop grossière pour effectuer des mesures fines sur les temps d'exécution des primitives du noyau. Le constructeur ne nous a jamais proposé d'autres outils de mesures. Aucune documentation n'est donnée sur les temps d'exécution des primitives SPART.

5.4 Conclusion

Ce travail a permis d'appréhender pleinement les problèmes de l'implantation du système réparti OMPHALE. Au cours de la phase de conception de l'architecture, ce travail a permis d'éclaircir certains points de l'architecture OMPHALE.
Le choix du développement du noyau NERF au dessus du noyau SPART est discutable en soi. On peut reprocher à cette solution sa trop grande rigidité. Les mécanismes SPART ne sont ni maitrisés ni

complétement mesurables.
Cette implantation est perfectible, mais elle a permis de répondre à la faisabilité de l'architecture.

Chapitre 6
Conclusion

Cette thèse a permis d'étudier pleinement les problèmes de l'implantation de l'architecture de système OMPHALE. Ce travail a débouché sur un premier prototype où les mécanismes suivants ont été réalisés:

- **Schéma d'exécution du module OMPHALE**
  Nous avons réalisé le schéma d'exécution du module OMPHALE tel qu'il a été proposé dans la thèse de Jean-Marc Geib. Un module OMPHALE est représenté par deux tâches SPART s'exécutant en parallèle. L'exécution du module est dirigée par les messages reçus.

- **Communication asynchrone par messages**
  Les modules OMPHALE communiquent et se synchronisent par un envoi asynchrone de messages. Le module OMPHALE prépare les messages dans sa zone d'émission. Cette zone d'émission est ensuite prise en charge par le système de transport qui assure le transfert des messages. Les messages en attente de traitement sont stockés dans la zone de réception du module. Le réseau est **transparent** c'est à dire que la communication entre modules voisins et distants se fait de la même manière.

- **Protection et nommage au travers de liens**
  Le lien permet à la fois le nommage des modules et la protection entre modules. Chaque module OMPHALE posséde un environnement où sont stockés ses liens. Cet environnement détermine la visibilité du module c'est à dire l'ensemble des modules accessibles. Cet ensemble évolue dynamiquement au cours de l'exécution du module.

- **Interface de programmation**
  Ce travail a permis d'établir précisément l'interface de programmation du noyau NERF. Le concept d'agence a permis de classifier les primitives et de définir très précisément les paramètres. L'interface doit être enrichie au niveau de la manipulation des liens. (Comparaison de deux liens).

- **Edition de liens dynamique**
  La résolution des références entre les objets se fait au cours de l'exécution de l'application. L'édition de liens dynamique sous OMPHALE peut se faire de deux manière:

  - Transfert de liens dans les messages
    Un message OMPHALE comprend à la fois les données que le programmeur peut directement modifier, mais aussi les liens qui accompagnent le message. Ces liens sont stockés dans une structure que le programmeur ne peut manipuler qu'à partir de primitives du noyau.

  - Serveur de noms symboliques (module système du CORTEX)
    Ce serveur catalogue un ensemble de noms globaux de modules.

- **Développement d'applications réparties**
  Un des aspects positifs de cette thèse est d'avoir pu mettre en oeuvre des applications réparties. Des études plus approfondies sur la programmation au dessus d' architectures réparties, mais ne rentrant pas dans le cadre de cette thèse seraient à mener. Autre question qu'on

pourrait se poser c'est de se demander sur quel type d'application, l'architecture OMPHALE serait parfaitement adaptée.

Des travaux approfondis mériteraint d'être entrepris pour faciliter le développement d'applications.

- Définition et implantation d'un langage de programmation de haut-niveau qui faciliterait le développement d'applications. Jean-Marie Place a réalisé un préprocesseur d'un langage OMPHALE autour de MODULA-2 [PLA 89].
- Définir des outils de mise au point d'applications en contexte réparti. De tels outils sont maintenant parfaitement maitrisés en contexte centralisé.

Pour faciliter la conception d'applications, il est nécessaire de définir complétement une couche CORTEX où l'on retrouve une grande multitude d'outils.

Chapitre 7
Bibliographie

[AFC 75] Groupe ALGOL
Manuel du langage algorithmique ALGOL 68
Herman, Paris (1975)

[ALB 85] A. Albot
SPART: Moniteur temps réel sur SPS7
Actes des journées SM90 Versailles Eyrolles pp 195-202 (1985)

[AMA 83] P. Amar
WINNIE un éditeur de textes multifeneêtres
Conférence BIGRE (1983)

[AND 83] F. André, D. Herman, J.P. Verjus
Synchronisation de programmes parallèles
Dunod Informatique (1984)

[BAC 57] J.W. Backus and all.
The FORTRAN Automatic Coding System
Proc. Western Joint Computer Reference pp 188-198 (1957)

[BAC 86] M.J. Bach
The design of the UNIX operating system
Prentice-Hall (1986)

[BAN 87] J.P. Banatre, M. Banatre, G. Muller et F. Ployette
Quelques aspects du système GOTHIC
Technique et Science Informatique vol 6 no 2 pp 170-174 (1987)

[BAS 77] F. Baskett, J.H. Howard, J.T. Montague
Task Communications in DEMOS
Proc. 6th Symposium Operating Systems Principles pp 23-31 (1977)

[BAU 84] C. Baudoin
Après Pascal: Modula-2
Technique et Science Informatique vol 3 no 6 pp 452-456 (1984).

[BET 85] C. Bétourné, X. Besnard and all
Modula-2 réparti sur SM90
Actes des journées SM90 Versailles Eyrolles pp 661-669 (1985)

[BIR 85] A.D. Birell
Implementing Remote Procedure Calls
Comm. ACM on Computer Systems, vol 3,1 pp 1-14 (1985)

[BIS 86] J.M. Bishop
Data Abstraction in Programming Languages
International Computer Science Series (1986)

[BOO 88] G. Booch
Ingénérie du logiciel avec ADA
texte français de Jean-Pierre Rosen
Addison-wesley Europe (1988)

[BOR 87] Compagnie BORLAND
Turbo C User's Guide
Documentation Technique Turbo C Borland (1987)

[BRO 82] H. Broze, A. Bruffaerts and all
Evaluation critique du système UNIX
Technique et Science Informatique vol 1 no 4 pp 315-324 (1982)

[BUL86a] Compagnie BULL
Structure générale du SPS7: Manuel de présentation
Documentation technique SPS7 (Avril 1986)

[BUL86b] Compagnie BULL
Système d'exploitation SPART: manuel de présentation
Documentation technique SPS7 (Avril 1986)

[BUL86c] Compagnie BULL
Système d'exploitation SPIX version 21.3
Documentation technique SPS7 (Avril 1986)

[CAR 82] T.A. Cargill
A robust distributed Solution to the Dining Philosophers problem
Software-Practice and Experience vol 12, 965-969 (1982)

[CHE 82] D.R. Cheriton
The Thoth System: Multi-process Structuring and Portability
The Computer Science Library. North-Holland (1982)

[CHE 83] D.R. Cheriton, W. Zwaenepoel
The Distributed V Kernel and its Performances for Diskless Stations
Proc. on Operating Systems Principle, Comm. ACM pp 129-140

[CHE 88] D.R. Cheriton
The V Distributed System
Communications of A.C.M. vol 31 no 3 pp 314-333 (1988)

[COM 83] D. Commer
Operating system design: the XINU approach
Prentice-Hall (1983)

[CON 63] M.E. Conway
Design of a Separable Transition Diagram Compiler
Comm. ACM vol 6,7 pp 396-408 (1963)

[COR 65] F.J. Corbato and all.
Introduction and Overview of the Multics system
Fall Joint Computer Conference pp 185-196 (1965)

[COR 69] F.J. Corbato
PL/I as a tool for system programming
Datamation vol. 15,5 pp 68-76 (1969)

[COR 81] CORNAFION (nom collectif)
Systèmes informatiques répartis, concepts et techniques
Dunod (1981)

[COU 71] P.J. Courtois
Concurrent control with readers and writers
Comm. ACM vol 14,10 pp 667-668 (1971)

{CRO 75] CROCUS (nom collectif)
Systèmes d'exploitation des ordinateurs
Dunod (1975)

[DEL 79] E. Delattre
Contribution à la répartition d'un système à structure de domaine sur un réseau local de micros-ordinateurs faiblement couplés
Thèse de Docteur-Ingénieur Université de Lille I (1979)

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

[DEL85b] E. Delattre, J.M. Geib, J.M. Place
Distributed Hardware for V.L.S.I. Component Sharing
Proc. Euromicro 85, Bruxelles (1985)

[DEL 86] E. Delattre, J.M. Place
Le support système des méthodologie de programmation orientée objet: 100 % d'"overhead"!
Journées AFCET GROPLAN (1986)

[DEL 88] E. Delattre, J.M. Geib, J.F. Mehaut, J.M. Place
Two implementations of the OMPHALE Distributed Architecture
IASTED International Symposium Applied Informatics (1988)

[DIJ 65] E.W. Diskstra
Co-operating Sequential Processes
in *Programming Languages* Genuys editor, London (1965)

[DIJ68a] E.W. Dijkstra
The strutcure of THE multiprogramming System
Communi of the ACM, vol 11, pp341-346 (1968)

[DIJ68b] E.W. Dijkstra
GOTO statement considered Harmfull
Comm ACM 11,3 pp147-148 (March 1968)

[DIJ 71] E.W. Dijkstra
Hierarchical ordering of Sequential Processes
Acta Informatica, vol 1,2 pp 115-138 (1971)

[DoD 83] D.o.D (U.S. Department of Defense)
Reference Manual for the Ada programming language
The Pentagon, Washington (1983)

[FAB 74] R.S. Fabry
Capability-based addressing
Comm, ACM vol 17,7 pp 403-412 (1974)

[FIN 81] U. Finger, G. Médigue
Architectures multi-processeurs et disponibilité: La SM90
L'echo des recherches no 105 pp15-21 (Juillet 1981)

[GAU 87] M.C. Gaudel, M. Soria, C. Froidevaux
Types de données et algorithmes
édité par l'I.N.R.I.A. collection didactique vol II (1987)

[GEI 89] J.M. Geib
L'architecture de système réparti OMPHALE
thèse LIFL, (Janvier 1989)

[GOL 83] A. Goldberg
Smalltalk-80: The Interactive Programming Environnement
Addison-Wesley (1983)

[GUI84a] M. Guillemont, H. Zimmermann, G. Morisset, J.S. Banino
CHORUS: Une architecture pour les systèmes répartis
Rapport de recherche INRIA 274 (1984)

[GUI84b] M. Guillemont
Etude comparative de quelques systèmes répartis
Technique et Science Informatique vol 3 no 1 pp 5-21 (1984)

[GUI 85] M. Guillemont, F. Armand, B Deslandes, M. Gien, P. Leonard
Vers une intégration CHORUS et UNIX
Actes des journées SM90 Versailles Eyrolles pp 465-477 (1985)

[HER 87] F. Hermann
CHORUS : Un environnement pour le développement et l'exécution d'applications réparties
Technique et Science Informatique vol 6 no 2 pp 162-165 (1987)

[HOA 75] C.A.R. Hoare
Monitors: an operating system structuring concept
Comm. ACM, vol 18,2 pp 95 (1975)

[HOL 83] R.C. Holt
Concurrent Euclid, the UNIX system, and Tunis
Addison-Wesley (1983)

[INM84a] Compagnie Inmos
OCCAM Programming Manual
Prentice-Hall Internationnal (1984)

[INM84b] Compagnie Inmos
IMS T424 Transputer Reference Manual
Documentation technique Inmos (1984)

[JON 79] A.K. Jones
The object model: a conceptual tool for structuring software

R. Bayer, R.M Graham and G. Seegmuller editors
Operating system: an advenced course, Springer-Verlag (1979)

[KAI 83] C. Kaiser, J.C. Hanout, H. Lucas, B. Martin
Expériences d'écriture et de transport de systèmes informatiques en Pascal
Technique et Science Informatique Vol 2 no 6 pp401-417 (1983)

[KER 78] B.W. Kerninghan, D.M. Ritchie
The C programming language
Prentice-Hall software series 1978

[KER 81] B.W. Kerningham
Why Pascal is not my Favorite Programming Language
Computing Science Technical
Report no 100 Bell Laboratories (1981)

[KNU 74] D.E. Knuth
Structured Programming with GOTO Statements
Computing Surveys vol 6 pp261-301 (1974)

[KRA 85] S. Krakowiak
Principe des systèmes d'exploitation des ordinateurs
Dunod (1985)

[KRA 87] S. Krakowiak
Systèmes d'exploitation répartis: progrès récents et tendance de la recherche
Technique et Science Informatique vol 6 (1987)

[LAM 78] L. Lamport
Time, Clocks and the ordering of events in a distributed system
Comm. ACM, vol 28,11 pp 125-133 (1978)

[LAN 87] G. Le Lann
Le projet Score: Les systèmes informatiques répartis temps réel
Technique et Science Informatique Vol 6 no 2 pp175-178 (1987)

[LEO 86] L. Leon
Programmation en Assembleur 68000
Eyrolles (1986)

[LIS 79] B. Liskov
Primitives for Distributed Computing

7th A.C.M Symp. on Operating System Principles pp 33-42 (1979)

[MET 76] R.M. Metcalfe, D.R. Boggs
Ethernet: Distributed Packet Switching for Local Computer Networks
Comm. A.C.M. vol 19,7 pp 395-404 (1976)

[MEY 80] B. Meyer, C. Baudoin
Méthodes de programmation
Eyrolles (1980)

[MOT 84] MOTOROLA
M68000, 16/32 bit microprocessor, programmer's reference manual
documentation technique MOTOROLA (1984)

[PAI 77] C. Pair, M.C. Gaudel
Les structures de données et leur représentation en mémoire
I.R.I.A. Rocquencourt (1977)

[PAP 88] M. Papageorgiou
Le système de gestion de fichiers répartis dans CHORUS
Technique et Science Informatique Vol 7 no 4 pp373-384 (1988)

[PET 85] J.L. Peterson, A. Silberschatz
Operating System Concepts
Addison-Wesley (1985)

[PLA 89] J.M. Place
Un pré-processeur Modula-2 pour OMPHALE
A Paraître

[RAS 84] R.F. Rashid
Accent : A communicayion oriented network operating system kernel
Proc Eight Symp on Operating System Principles
ACM Operating System review vol 17,5 pp 64-75 (1974)

[RAY 84] M. Raynal
Algorithmique du parallélisme
Dunod Informatique (1984)

[RIT 74] D.M. Ritchie, K. Thompson
The UNIX Time-Sharing System
Communications of the ACM vol 17 no 7 pp365-375 (1974)

[SAA 70] H.J. Saal, W.E. Riddle

Communicating semaphores
SLAC, CGTM 117 Stanford (1970)

[SCE 82] Bureau d'orientation de la Normalisation en Informatique
Rapport Sceptre
Rapport BNI no 16/2 Septembre (1982)

[STR 81] B. Stroustrup
Long Return: A technique for improving the Efficiency of Inter-module communication
Software-practice and Experience, vol 11 pp 131-143 (1981)

[STR 86] B. Stroustrup
The C++ Programming Language
Addison-Wesley (1986)

[SUN 86] Sun Microsystems, inc
Networking guide: IPC primer
documentation technique SUN (1986)

[TAN 87] A.S. Tanenbaum
Operating Systems: Design and implementation
Prentice-Hall (1987)

[VOJ 84] D. Vojnovic, F. Browaeys, H. Derriennic, P. Desclaud and all.
Sceptre : proposition de noyau normalisé pour les exécutifs temps-réel)
Technique et Science Informatique Vol 3 no 1 pp45-62 (1984)

[WIR 71] N. Wirth
The Programming Language PASCAL
Acta Informatica 1,1 pp35-63 Springer-Verlag (1971)

[WIR 83] N. Wirth
Programming in Modula-2
Springer-Verlag (1983)

Chapitre 8
Annexes

Annexe A
Les agences SPART

A.1 Déclarations communes

```
#define    LOCAL              0
/* l'objet à creer est un objet local */

#define    GLOBAL             0x8000
/* l'objet à creer est un objet global */

#define    INFINITE           0xffffffff
/* la primitive est blocante. le blocage peut
   durer un temps indeterminé                */


typedef unsigned short xx_id ;
/*identificateur d'objet */

typedef unsigned int status ;
/* status rendu par les primitives agences */


/* Definition d'une adresse generalisee
   ------------------------------------ */


typedef struct {
                xx_id seg_id ;
                /* nom systeme de l'objet segment */

                unsigned int offset ;
                /* deplacement dans le segment */
               } general_address ;

/* les codes d'erreur */

#define OK         0
/* La primitive s'est déroulée sans problème */

#define  E_UNKNOWN_OBJ      1
/* L'identificateur d'objet est erroné */
```

```
#define E_OBJ_OVERFLOW      2
/* La création d'objet est impossible. Les tables d'objet du
   système est saturée */
```

A.2 Agence de gestion des programmes

```
#define   REENTRANT                    0
#define   NOT_REENTRANT      1

/* numero de l'unite disque sur laquel se trouve l'image memoire */
#define   SYST_UNIT           -1


status load (unit_num,file_name,prog_type,prog_id)
char unit_num ; /* numéro de l'unité disque contenant l'image executable
*/
char *file_name ; /* nom du fichier UNIX contenant l'image executable */
unsigned short prog_type ; /* programme REENTRANT ou pas */
xx_id *prog_id     ;
/* chargement d'une image mémoire executable */

status unload (prog_id)
xx_id prog_id ;
/* Vidage d'un programme. La destruction n'est effective qu'à la
fin de toutes les tâches créees sur ce programme                */

/*****************************************/
/* erreurs de l'agence programme         */
/*****************************************/

#define E_INEX_FILE                    0xa0
/* load : fichier inexistant                              */

#define E_INVALID_PROG      0xa1
/* load : programme invalide                              */

#define E_IO       0xa2
/* load : erreur entree sortie physique                    */

#define E_IO_TIMOUT         0xa3
/* load : non réponse du module d'échange      */
```

A.3 Agence de gestion des tâches

```
#define  USR                0x0
#define  SYS                0x2000

/* Definition de l'etat d'une tache */

#define  PRETE              0
#define  EN_COURS             PRETE+1
#define  EN_ATTENTE         EN_COURS+1
#define  HORS_SERVICE       EN_ATTENTE+1

typedef struct
             {
              xx_id task_id ;
               /* nom systeme de l'objet tache */

              xx_id prog_id ;
               /* nom systeme de l'objet programme */

              char prio ;
               /* priorite de la tache */

              char state ;
               /* etat de la tache */

              short task_type ;
               /* type de la tache USR ou SYST */

              char ut_num ;
               /* numero de l'ut sur laquel s'execute la tache */

              char nb_ut ;
               /* nombre d'ut de la configuration */

              short env_size ;
               /* taille de l'environnement */
             } task_param ;

status create_task (
                    prog_id ,
                    priority ,
                    privilege ,
                    max_resume ,
```

```
                    task_id
                    )
xx_id prog_id ;
short priority, privilege, max_resume ;
xx_id *task_id ;

status start_task (
                    task_id,
                    mess
                    )
xx_id task_id ;
char  *mess ;

exit_task ()

status kill_task (
                   task_id
                   )
xx_id task_id ;

status suspend_task ()

status resume_task (
                    task_id
                    )
xx_id    task_id ;

status get_task_param (
                       task_id,
                       p_task_param
                      )
xx_id task_id ;
task_param *p_task_param ;

status wait_end_task (
                      task_id,
                      delay
                      )
xx_id task_id      ;
unsigned int delay          ;

/*********************************/
/* erreurs renvoyées             */
/*********************************/
```

```
#define  E_USED_PROG        0x20
/* create_task : programme non reentrant en cours
                 d'utilisation    */
#define  E_TASK_STAT        0x21
/* etat de tache incorrect */

#define  E_RESUME_OVERFLOW  0x22
/* saturation compteur d'activation */

#define  E_PRIVILEGE        0x23
/* create_task : creation d'une tache en mode systeme
                 par une tache en mode user       */

#define  E_OWNER   0x24
/* attendre sa propre fin */

#define  E_TASK_WAIT        0x25
/* il y a deja une tache en attente */

#define  W_TASK_RUNNING     0x26
/* delay d'attente nul */
```

A.4 Agence de gestion mémoire

```
#define READ_ONLY  0xc
#define READ_WRITE          0x8

status create_segment (
                       attribute, /* LOCAL ou GLOBAL */
                       length,
                       seg_id
                      )
unsigned short attribute ;
unsigned int   length ;
xx_id         *seg_id ;
/* creation d'un segment en memoire locale ou globale */

status access_segment (seg_id,
                       access_type,
                       logical_base_adr
                      )
xx_id              seg_id ;
unsigned short     access_type ;
unsigned int       *logical_base_adr ;
/* Attachement du segment à une adresse choisie par SPART */

status deaccess_segment (seg_id)
xx_id    seg_id ;
/* detachement du segment */

status delete_segment (seg_id)
xx_id    seg_id ;
/* destruction du segment seg_id */


status declare_segment (seg_id,
                        granule_length
                       )
xx_id              seg_id ;
unsigned short     granule_length ;
/* Initialisation de l'algorithme d'allocation dans le segment.
   Le segment sera découpé en granule de la longueur specifiée
*/

status alloc (seg_id,
              block_length,
```

```
              block_adr
            )
xx_id               seg_id ;
unsigned int        block_length ;
unsigned int        *block_adr ;
/* Allocation d'un bloc de longueur block_length
   dans le segment seg_id
*/

status free (seg_id,
             block_adr,
             block_length
           )
xx_id               seg_id ;
unsigned int        block_adr ;
unsigned int        block_length ;

status map_segment (seg_id,
                    access_type, /* READ_ONLY, READ_WRITE */
                    logical_base_adr
                   )
xx_id               seg_id ;
unsigned short      access_type ;
unsigned int        logical_base_adr ;

status gen_addr (logical_adress,
                 general_adress
                )
unsigned int         logical_address  ;
general_address     *general_addr     ;


status loc_addr (general_addr,
                 logical_addr
                )
general_address *general_addr        ;
unsigned int    *logical_address ;

/*****************************************************/
/* Codes erreurs spécifiques à l'agence mémoire */
/*****************************************************/

#define E_PHYS_SPACE        0x30
/* create_segment : saturation memoire physique du système */
```

```
#define  E_LOGIC_SPACE      0x31
/* access_segment : l'espace logique de l'espace utilisateur est
                    saturé                                      */

#define  E_SEG     0x32
/* declare_segment : Le segment est déjà initialisé */


#define  E_ACCESS 0x34
/* alloc_memory, declare_segment, free_memory, loc_addr
   Le segment n'est pas mappé (access_segment ou map_segment)
   dans l'espace d'adressage utilisateur */

#define  E_NO_SPACE         0x35
/* alloc_memory : il n'y a pas de bloc libre dans le segment     */

#define  E_FREE    0x36
/* free_memory : Le bloc n'appartient pas au segment ou
                 a déjà été libéré */

#define E_ACCESS_TYPE       0x37
/* declare_segment, alloc_memory ou free sur un segment
   en lecture simple (READ_ONLY)                                 */

#define  E_SIZE    0x38
/* create_segment : demande de creation d'un segment
                    de 0 Koctet                          */

#define  E_GRANULE_LENGTH   0x39
/* declare_segment : taille du granule d'allocation
                     superieure à la taille du segment  */

#define  E_ACCESSED_SEG     0x3a
/* segment deja accede par la tache  */

#define E_OFFSET   0x3b
/* deplacement dans le segment invalide       */

#define E_LOGIC_ADDR        0x3c
/* adresse logique invalide                   */
```

A.5 Agence événement

```
#define SET          1  /* evenement positionne         */
#define NOT_SET      0  /* evenement non positionne */

/* type evenement */

typedef enum { EVT0,EVT1,EVT2,EVT3,EVT4,EVT5,EVT6,EVT7,EVT8,EVT9,
               EVT10,EVT11,EVT12,EVT13,EVT14,EVT15
             } event_type ;

/* type liste d'evenement */

#define  LEVT0     1<<0
#define  LEVT1     1<<1
#define  LEVT2     1<<2
#define  LEVT3     1<<3
#define  LEVT4     1<<4
#define  LEVT5     1<<5
#define  LEVT6     1<<6
#define  LEVT7     1<<7
#define  LEVT8     1<<8
#define  LEVT9     1<<9
#define  LEVT10    1<<10
#define  LEVT11    1<<11
#define  LEVT12    1<<12
#define  LEVT13    1<<13
#define  LEVT14    1<<14
#define  LEVT15    1<<15

typedef unsigned short event_list_type ;

status test_event (event,
                   event_state
                  )
event_type event ;
unsigned short *event_state ;
status set_event (event,
                  task_id
                 )
event_type event ;
xx_id task_id ;

status wait (liste)
```

```
event_list_type liste ;

/*******************************************************/
/* Codes erreurs specifiques à l'agence événement  */
/*******************************************************/

#define  E_EVENT  0x90
/* numero d'evenement incorrect */

#define E_EVENT_LIST        0x91
/* liste d'evenements incorrecte */
```

A.6 Agence de gestion des sémaphores

```
status s_request (
                  sem_id,
                  delay
                 )
xx_id       sem_id ;
unsigned int delay ;
/* acces partage au semaphore sem_id */

status e_request (
                  sem_id,
                  delay
                 )
xx_id        sem_id ;
unsigned int  delay ;
/* acces exclusif au semaphore sem_id */


status release (
                sem_id
               )
xx_id sem_id ;
/* libération du sémaphore : primitive V */

/*********************************************************/
/* Codes d'erreur spécifiques à l'agence sémaphore  */
/*********************************************************/

#define  W_USED_SEM         0x50
/* delete_semaphore : Le sémaphore était en cours d' utilisation */

#define E_DELETED_SEM       0x51
/* s_request, e_request : Le sémaphore a été détruit au cours
                           de l'attente */

#define E_REL_SEM  0x52
/* release : Le sémaphore est déjà libre */

#define E_USED_SEM          0x53
/* s_request, e_request : Le sémaphore n'est pas libre et on
                          ne veut pas d'attente */
```

A.7 Agence de gestion des messages

```
#define MESS_LENGTH         8         /* longueur message SPART */


status create_mbx (
                    attribute, /* LOCAL ou GLOBAL */
                    message_nb, /* nombre de messages de la boite à lettres
*/
                    mbx_id
                   )
unsigned short      attribute;
unsigned short      message_nb;
xx_id               *mbx_id;
/* creation de la boite à lettres mbx_id */

status delete_mbx (
                    mbx_id
                   )
xx_id mbx_id ;
/* detruire la boite à lettres mbx_id */


status send (
              mess,
              mbx_id,
              delay
             )
char         mess[MESS_LENGTH];
xx_id        mbx_id;
unsigned int delay;
/* envoi du message mess vers la boite à lettres mbx_id */

status receive (
                 mess,
                 mbx_id,
                 delay
                )
char         mess[MESS_LENGTH];
xx_id        mbx_id;
unsigned int delay;

/*****************************************************/
/* Code erreurs spécifiques à l'agence Message    */
```

```
/*****************************************************/

#define  W_USED_MBX
/* delete_mbx : messages ou tâches en attente sur la mailbox */

#define E_DELETED_MBX       0x41
/* receive :  la mailbox a été détruite */

#define E_MBX_FULL          0x42
/* send : la mailbox est pleine */

#define E_MBX_EMPTY         0x43
/* receive : la mailbox est pleine */
```

Annexe B
Le contexte standard d'une tâche SPART

```
typedef struct {
              lien_2 lien ;
               /* lien de chainage dans les files de */
               /* l'ordonnanceur                     */

              char prio ;
               /* priorite de la tache        */

              char etat ;
               /* etat de la tache (hors-service,prete */
              /* en attente,zombie,...)                 */

               unsigned int evt_att ;
               /* evenements attendus */

              unsigned int evt_arr ;
               /* evenements arrives */

              save_reg *ssp ;
               /* pointeur pile systeme */

              xx_id task_id ;
               /* nom systeme de la tache */

              char pres_ugm ;
               /* tache presente dans l'ugm */

              char num_desc_ugm ;
               /* numero de descripteur ugm */

              unsigned char num_der_reg_ugm ;
              /* numero dernier registre ugm   */
               /* valide dans l'espace de la    */
               /* tache                         */

              char fpu_saved ;
               /* fpu registers already saved in system call */

              ugm_desc *ugm_s_adr ;
              /* adresse de la table de sauvegarde */
```

```
/*  des registres ugm de la tache          */

 short bamax ;
 /* sauvegarde du registre base/longueur */

char kill_d ;
 /* meurtre de la tache demande */

 char kill_i ;
 /* meurtre de la tache interdit */

 char          *u ;
 /* pointeur structure U pour FMS  */

xx_id session_id ;
/* num objet session_debug */

 unsigned int db_ctx;
/* ptr contexte debug */

unsigned int syst_be;
/* adresse reprise bus error systeme */

unsigned short sr ;
 /* privilege de la tache */

 short max_resume ;
 /* nombre maximum de reactivation */

short cpt_resume ;
 /* compteur de reactivation */

xx_id stack_id ;
 /* nom systeme du segment pile user       */

xx_id data_id ;
 /* nom systeme du segment data */

 xx_id delay_id ;
/* ident du delay pour time-out */

head_2 iocb_head ;
/* tete de chainage des iocb */
```

```
        int ex_adr;
        /* adresse handler d'exception */

        union fpu_ctx *fpu_ctx;
        /* address of fpu save area */

        char flag;
        char ruf;

        char start_msg[8];
        /* initial message */

        char spart_reserved[12];
        /* reserve pour extensions futures */

    } ctxt ;
```

Annexe C
Spécification des primitives du noyau

C.1 Spécification des primitives de l'agence liens

NAME
DuplicateLink : Cette primitive duplique un lien de l'environnement.

SYNOPSIS

en C

```
Status DuplicateLink (IdLien1,IdLien2)
     unsigned int IdLien1,*IdLien2 ;
```

en Modula-2

```
PROCEDURE DUPLICATELINK
            (    IdLien1   : ModuleId ;
             VAR IdLien2   : ModuleId
            ) : Status ;
```

DESCRIPTION

Cette primitive duplique le lien *IdLien1*. *IdLien2* est l'identificateur de la copie du lien.

ERRORS

- BADIDLINK: L'identificateur de lien est erroné.
- CANNOTDUPLICATETHISLINK: Ce lien n'est pas duplicable.
- LINKALREADYINBUFFER: La duplication d'un lien se trouvant dans un message est prohibée.
- NOSPACEINENVIRONNEMENT: L'environnement du module est saturé.

SEE ALSO
DestroyLink, RetrieveLink.

NAME

DestroyLink : destruction d'un lien de l'environnement.

SYNOPSIS

en C

```
Status DestroyLink (M)
                   unsigned int M ;
```

en Modula-2

```
PROCEDURE DESTROYLINK (M: ModuleId) : Status ;
```

DESCRIPTION

Cette primitive détruit le lien *M* de l'environnement. Au cours d'une phase d'activité, un lien détruit par la primitive *DestroyLink* peut être récupéré par la primitive *DestroyLink.*

ERRORS

- BADIDLINK : L'identificateur *M* est invalide.
- LINKALREADYDELETED : Le lien *M* est déjà détruit .
- LINKALREADYINBUFFER: La destruction d'un lien se trouvant dans un message n'est pas possible.
- CANNOTDELETETHISLINK : La destruction de ce lien est interdite. Cette erreur est retournée à la destruction de liens permanents.

SEE ALSO

DuplicateLink, RetrieveLink.

NAME

RetrieveLink: Cette primitive récupère un lien dont la destruction avait été demandée.

SYNOPSIS

en C

```
Status RetrieveLink (IdLien)
                    unsigned int IdLien ;
```

en Modula-2

```
PROCEDURE DESTROYLINK (M: ModuleId) : Status ;
```

DESCRIPTION

Cette primitive annule l'effet de destruction qui avait été demandée.

ERRORS

- BADIDLINK : L'identificateur *IdLien* est invalide.
- LINKWASNOTDELETED : La destruction de ce lien n'avait pas été demandée.

SEE ALSO

DuplicateLink, DestroyLink.

NAME

TransferLink : Cette primitive place un lien dans un message.

SYNOPSIS

en C

```
Status TransferLink (IdLien,IdMessage)
       unsigned int IdLien ;
       unsigned short int IdMessage ;
```

en Modula-2

```
PROCEDURE TRANSFERLINK
           (    IdMessage : MessageId ;
                IdLien : ModuleId
           ) : Status ;
```

DESCRIPTION

Le lien *IdLien* est placé dans le message *IdMessage*.

ERRORS

- BADIDMESSAGE: L'identificateur *IdMessage* est invalide.
- BADIDLINK: L'identificateur *IdLien* est invalide.
- LINKALREADYINBUFFER: Le lien se trouve déjà dans un message.
- CANNOTTRANSFERTHISLINK: Il n'est pas possible de transférer un lien Permanent.
- LINKDELETED: Le lien a été préalablement détruit.
- NOSPACESEGSYSGLOBAL: La mémoire du système est saturée

SEE ALSO

CreateMessage, DeleteMessage, TakeAgainLink.

NAME

TakeAgainLink : Reprendre un lien d'un message.

SYNOPSIS

en C

```
Status TakeAgainLink (IdMessage,IdLien)
                     unsigned int IdLien ;
                     unsigned short int IdMessage ;
```

en Modula-2

```
PROCEDURE TAKEAGAINLINK (IdMessage : MessageId ;
                         IdLien  : ModuleId
                        ) : Status ;
```

DESCRIPTION

Il s'agit de reprendre du message *IdMessage* le lien *IdMessage*.

ERRORS

- BADIDMESSAGE: *IdMessage* est invalide.
- BADIDLINK: *IdLien* est invalide.
- LINKNOTINMESSAGE: Le lien *IdLien* n'est pas dans le message *IdMessage*.

SEE ALSO

TransferLink, CreateMessage.

NAME

SetNoUse : Cette primitive restreint l'ensemble des services accessibles avec un lien.

SYNOPSIS

en C

```
Status SetNoUse (M,Services)
       unsigned int M ;
       unsigned short int Services ;
```

en Modula-2

```
PROCEDURE SETNOUSE ( M : ModuleId ;
                     Services : BITSET) : Status ;
```

DESCRIPTION

Cette primitive restreint les services accessibles avec le lien *M* avec le paramètre *Services.*

ERRORS

- BADIDLINK : Identificateur de lien incorrect.
- LINKALREADYINBUFFER : Ce lien a été transféré dans un lien. Ses attributs ne sont donc pas modifiables.
- LINKDELETED : Ce lien a été détruit.
- CANNOTMODIFYTHISLINK : Vous ne pouvez modifier les attributs de ce lien (Pour les liens Permanents)

SEE ALSO

SetOneUse, SetNoDups.

NAME

SetOneUse : Pour les services utilisables d'un lien, cette primitive réduit le nombre d'utilisation à une seule.

SYNOPSIS

en C

```
Status SetOneUse (M,Services)
       unsigned int M ;
       unsigned short int Services ;
```

en Modula-2

```
PROCEDURE SETONEUSE ( M : ModuleId ;
                      Services : BITSET) : Status ;
```

DESCRIPTION

Les *Services*, si ils sont utilisables par le lien *M* ne seront plus utilisables qu'une seule fois.

ERRORS

- BADIDLINK : Identificateur de lien incorrect.
- LINKALREADYINBUFFER : Ce lien a été transféré dans un lien. Ses attributs ne sont donc pas modifiables.
- LINKDELETED : Ce lien a été détruit.
- CANNOTMODIFYTHISLINK : Vous ne pouvez modifier les attributs de ce lien (Pour les liens Permanents)

SEE ALSO

SetNoUse, SetNoDups.

NAME

SetNoDups : Cette primitive rend un lien non duplicable.

SYNOPSIS

en C

```
Status SetNoDups (M)
       unsigned int M ;
```

en Modula-2

```
PROCEDURE SETNODUPS ( M : ModuleId) : Status ;
```

DESCRIPTION

Le Lien *M* ne pourra plus être dupliqué. L'utilisation de la primitive *DuplicateLink* sera prohibée.

ERRORS

- BADIDLINK : Identificateur de lien incorrect.
- LINKALREADYINBUFFER : Ce lien a été transféré dans un lien. Ses attributs ne sont donc pas modifiables.
- LINKDELETED : Ce lien a été détruit.
- CANNOTMODIFYTHISLINK : Vous ne pouvez modifier les attributs de ce lien (Pour les liens Permanents)

SEE ALSO

SetNoUse, SetOneUse.

NAME

StateLink : Cette primitive retourne les attributs d'un lien.

SYNOPSIS

en C

```
Status StateLink (IdLien, UseOrNo, OneUse, Duplicate)
       unsigned int IdLien ;
       unsigned short int *UseOrNo, *OneUse ;
       unsigned short int Duplicate ;
```

en Modula-2

```
PROCEDURE STATELINK (M : ModuleId ;
                     VAR UseOrNo, OneUse : BITSET ;
                     VAR Duplicate : BOOLEAN ) : Status ;
```

DESCRIPTION

La primitive retourne dans *UseOrNo* et *OneUse* les attributs du lien *M*.

ERRORS

- BADIDLINK : Identificateur de lien incorrect.
- LINKDELETED : Ce lien a été détruit.
- LINKALREADYBUFFER : Ce lien a été tranféré dans un message.

NAME

NumberLinkReceived: Cette primitive retourne le nombre de liens se trouvant dans le dernier message reçu par le module (Au cours de la présente phase d'activité).

SYNOPSIS

en C

```
Status NumberLinkReceived (n)
       int *n ;
```

en Modula-2

```
PROCEDURE NUMBERLINKRECEIVED
            (VAR n : LONGINT
            ) : Status ;
```

DESCRIPTION

n contiendra le nombre de liens reçus dans le dernier message.

ERRORS

- Cette primitive ne renvoie jamais d'erreur.

SEE ALSO

IdLinkReceived, Wait.

NAME

IdLinkReceived: Obtenir l'identificateur de lien placé dans un message.

SYNOPSIS

en C

```
Status IdLinkReceived (X,M)
     integer X ;
     unsigned int M ;
```

en Modula-2

```
PROCEDURE IDLINKRECEIVED (X : LONGINT ;
                          VAR M : ModuleId) : Status ;
```

DESCRIPTION

ERRORS

- ERRORNUMLINK: Le rang X du lien est incorrect.

SEE ALSO

NumberLinkReceived, Wait.

C.2 Spécification des primitives de l'agence Messages

NAME

CreateMessage : C'est la primitive de création d'un message OMPHALE. Le message est placé dans la zone d'émission du module.

SYNOPSIS

en C

```
Status CreateMessage (IdLien,IdQuai,mess,l,IdMessage)
       unsigned int IdLien ;
       unsigned short IdQuai,*IdMessage,l ;
       char *mess ;
```

en Modula-2

```
PROCEDURE  CREATEMESSAGE
           (    IdLien   : ModuleId;
                IdQuai   : ServiceId;
                Longueur : CARDINAL ;
            VAR mess     : ARRAY OF CHAR;
            VAR IdMessage: MessageId
           ) : Status ;
```

DESCRIPTION

CreateMessage crée un nouveau message dans la zone d'émission (ZE). Le message est émis vers le module pointé par *IdLien* sur le quai *IdQuai*. Le message est identifié par *IdMessage*.

ERRORS

- BADIDLINK : L'identificateur de lien *IdLien* est invalide.
- LINKDELETED : *IdLien* a été détruit.
- LINKALREADYINBUFFER : Le lien a été transféré dans un message.
- NOMOREACCESS : Le service *IdQuai* n'est pas accessible par ce lien.
- NOSPACESEGSYSGLOBAL : Le segment système du site est saturé

SEE ALSO

DeleteMessage, DeleteAllMessages, TransferLink, TakeAgainLink.

NAME

DeleteMessage : Destruction d'un message de la zone d'émission.

SYNOPSIS

en C

```
Status DeleteMessage (IdMessage)
       unsigned short int IdMessage ;
```

en Modula-2

```
PROCEDURE DELETEMESSAGE (IdMessage : MessageId) : Status ;
```

DESCRIPTION

Le message repéré par *IdMessage* sera détruit de la zone d'émission. Les liens se trouvant dans le message sont automatiquement replacés dans l'environnement.

ERRORS

- BADIDMESSAGE : L'identificateur de message *IdMessage* est invalide.

SEE ALSO

CreateMessage, DeleteAllMessages.

NAME

DeleteAllMessages : Cette primitive détruit l'ensemble des messages OMPHALE de la zone d'émission du module.

SYNOPSIS

en C

```
Status DeleteAllMessages ()
```

en Modula-2

```
PROCEDURE DELETEALLMESSAGES () : Status ;
```

DESCRIPTION

Les messages de la zone d'émission sont détruits. Les liens se trouvant dans les messages sont replacés dans l'environnement.

ERRORS

- Aucune erreur n'est retournée par cette primitive.

SEE ALSO

CreateMessage, DeleteAllMessages.

NAME

SetServ : Cette primitive précise les quais à ouvrir pour la prochaine phase d'activité du module.

SYNOPSIS

en C

```
Status SetServ (Services)
unsigned short int Services ;
```

en Modula-2

```
PROCEDURE SETSERV (Services : BITSET) : Status ;
```

DESCRIPTION

SetServ indique les quais qui sont ouverts à la prochaine phase d'activité.

ERRORS

- Aucune erreur n'est retournée par cette primitive.

SEE ALSO

PutServ, ResServ, ReadServ.

NAME

PutServ : Cette primitive ajoute aux quais ouverts les *Services* passés an paramétre.

SYNOPSIS

en C

```
Status PutServ (Services)
unsigned short int Services ;
```

en Modula-2

```
PROCEDURE PUTSERV (Services : BITSET) : Status ;
```

DESCRIPTION

PutServ ajoute aux quais déjà ouverts les *Services* passés en paramétres. Il y a un ou logique qui est fait entre les quais ouverts et les services à ouvrir.

ERRORS

- Aucune erreur n'est retournée par cette primitive.

NAME

ResServ : Cette primitive ferme les quais qui sont spécifiés.

SYNOPSIS

en C

```
Status ResServ (Services)
unsigned short int Services ;
```

en Modula-2

```
PROCEDURE RESSERV (Services : BITSET) : Status ;
```

DESCRIPTION

Les quais *Services* sont fermés. Aucun message ne peut être choisi par le noyau pour la prochaine phase d'activité.

ERRORS

- Aucune erreur n'est retournée par cette primitive.

SEE ALSO

SetServ, ResServ, ReadServ.

NAME

SizeMessage : Cette primitive retourne la taille en octets du dernier message reçu. (Sans compter les liens).

SYNOPSIS

en C

```
Status SizeMessage ()
```

en Modula-2

```
PROCEDURE SIZEMESSAGE () : INTEGER ;
```

DESCRIPTION

Cette primitive retourne la taille (en octets) du dernier message reçu. La primitive *LastMessage* permet d'accéder au contenu de ce message.

ERRORS

- Aucune erreur n'est retournée par cette primitive.

SEE ALSO

LastMessage.

NAME

LastMessage : Cette primitive renvoie le contenu du dernier message. Le programmeur doit fournir l'adresse d' une zone mémoire de taille suffisante pour contenir ce message.

SYNOPSIS

en C

```
Status LastMessage (ch)
char *ch ;
```

en Modula-2

```
PROCEDURE LASTMESSAGE (ch : ADDRESS) ;
```

DESCRIPTION

Cette primitive retourne le contenu du dernier message reçu. Le programmeur doit fournir au noyau une zone mémoire pour y placer le contenu.

ERRORS

- Aucune erreur n'est retournée par cette primitive.

SEE ALSO

SizeMessage.

NAME

Wait : L'appel de cette primitive provoque la fin de la phase d'activité du module OMPHALE. Le module sera réveillé à la réception d'un message sur un quai ouvert.

SYNOPSIS

en C

```
Status Wait ()
```

en Modula-2

```
PROCEDURE WAIT () : ServiceId ;
```

DESCRIPTION

Le numéro du prochain service à executer est renvoyé par la primitive *Wait*. Les primitives *LastMessage* et *SizeMessage* permettent d'accéder au dernier message reçu. Les primitives *IdLinkReceived* et *NumberLinkReceived* retournent les liens qui se trouvaient dans ce dernier message.

NAME

InitBuffer : Initialisation de la zone d'emission (ZE).

SYNOPSIS

en C

```
Status InitBuffer ()
```

en Modula-2

```
PROCEDURE INITBUFFER () : Status ;
```

DESCRIPTION

Cette fonction initialise les données du contexte du module OMPHALE. Cette primitive doit être appelée au début de la première phase d'activité du module.

ERROR

- Aucune erreur n'est renvoyée par cette primitive.

Annexe D
L'interface Modula-2 au noyau NERF

```
DEFINITION MODULE Omphale ;

    FROM Error   IMPORT Status ;
    FROM ModulaK IMPORT LONGINT ;
    FROM SYSTEM  IMPORT ADDRESS ;

    EXPORT QUALIFIED

                Reponse, Null,
                Myself, ServeurNom, ServeurModule,

                ModuleId, ServiceId, MessageId, NomModule,

                INITBUFFER,

                SETSERV, RESSERV, PUTSERV,
                READSERV,

                WAITMESSAGE, LASTMESSAGE,
                SIZEMESSAGE,

                CREATEMODULE,

                CREATEMESSAGE,  DELETEMESSAGE,

                RETRIEVELINK,
                DESTROYLINK, TRANSFERLINK, DUPLICATELINK,
                READLINK ,
                TAKEAGAINLINK,
                NUMBERLINKRECEIVED, IDLINKRECEIVED ,
                SETNODUPS, SETONEUSE, SETNOUSE, STATELINK  ;

CONST
   Myself       = 0  ;   (* ModuleId d'un module vers lui-meme *)
   ServeurNom   = 1  ;   (* ModuleId du serveur de nom vers lui-meme *)
   ServeurModule = 2  ;   (* ModuleId du serverur de module *)
   Null         = 32766 ; (* ModuleId null absence de lien  *)
   Reponse      = 0  ;   (* No de quai reserve pour les reponses au RemoteCall *)

TYPE
   ModuleId  = LONGINT ;              (* TYPE 'Nom local de module      *)
   ServiceId = [0..15] ;              (* TYPE 'No de Quai'               *)
   MessageId = CARDINAL ;             (* TYPE 'Nom de message envoye     *)
   NomModule = ARRAY [0..7] OF CHAR ; (* TYPE 'Nom de prototypé          *)

(* Procedure d'initialisation du module      *)
(* cette primitive doit etre appelee au      *)
(* debut de la premiere d'activite du module *)
```

```
PROCEDURE INITBUFFER () : Status ;
(* Initialisation de la ZE          *)


(***********************************************)
(* Creation d'un module a partir d'un prototype *)
(***********************************************)

PROCEDURE CREATEMODULE (Nom : NomModule; VAR M : ModuleId) : Status ;

(***********************************************)
(* Operations systemes internes au module       *)
(***********************************************)

PROCEDURE WAITMESSAGE () : ServiceId ;
       (* Primitive WAIT *)

PROCEDURE LASTMESSAGE (m : ADDRESS) ;
(* message recu au cours de cette phase d'activite *)

PROCEDURE SIZEMESSAGE () : INTEGER ;
(* renvoie la taille du message recu *)

(***************************************)
(* Procedures de validation des quais:  *)
(* parametre BITSET= 1 Bit par quai     *)
(***************************************)

PROCEDURE SETSERV (CFG : BITSET) ;   (* 1= Validation,   0= No change    *)
PROCEDURE RESSERV (CFG : BITSET) ;   (* 1= Invalidation, 0= No Change    *)
PROCEDURE PUTSERV (CFG : BITSET) ;   (* 1= Validation,   0= Invalidation *)

PROCEDURE READSERV (VAR CFG : BITSET) ;

PROCEDURE NUMBERLINKRECEIVED (VAR nb : LONGINT) : Status ;

PROCEDURE IDLINKRECEIVED (X : CARDINAL ;
                          VAR M : ModuleId) : Status ;

PROCEDURE TRANSFERLINK           (* Recopie d un lien dans un message *)
            (    M : ModuleId ;
                 X : MessageId)
                           : Status ;

PROCEDURE TAKEAGAINLINK (Mes : MessageId ;
                         Mod : ModuleId) : Status ;

PROCEDURE  CREATEMESSAGE     (* Envoi d un message                *)
          (    M: ModuleId;  (* Nom local du module destinataire  *)
               S: ServiceId; (* Nom du service destinataire       *)
           VAR X: MessageId; (* no du message cree                *)
               L: CARDINAL;
            Mess:ADDRESS     (* Contenu du message hors liens     *)
            ) : Status ;
```

```
PROCEDURE DELETEMESSAGE (X: MessageId) : Status ;
(* Destruction d'un message          *)

PROCEDURE DELETEALLMESSAGES () : Status ;
(* Destruction de tous les messages de la zone d'emission *)


PROCEDURE READMESSAGE (    X: MessageId ;
                       VAR M: ARRAY OF CHAR
                      ) : Status ;

(***************************************)
(* Gestion des liens                   *)
(***************************************)

PROCEDURE DESTROYLINK (M: ModuleId) : Status ;
(* Enlever un lien de l environnement *)

PROCEDURE RETRIEVELINK (M: ModuleId) : Status ;

PROCEDURE DUPLICATELINK
            (    Original: ModuleId ;      (* Nom local a dupliquer              *)
             VAR Copie: ModuleId)          (* Nom local recupere                 *)
                              : Status ;

PROCEDURE READLINK  (    M : ModuleId ;
                     VAR Use,NumberUse : BITSET
                    ) : BITSET ;

(***************************************)
(* Gestion des droits des liens        *)
(* Parametres:                         *)
(*   M: Nom local du module (lien)     *)
(*   S: Liste des quais concernes      *)
(*      (1: Concerne, 0: No Change)    *)
(***************************************)

PROCEDURE SETONEUSE (M: ModuleId; S: BITSET) : Status ;
(* Une seule utilisation *)

PROCEDURE SETNOUSE (M: ModuleId; S: BITSET) : Status ;
(* Aucune utilisation    *)

PROCEDURE SETNODUPS (M: ModuleId) : Status ;
(* Duplication du lien interdite   *)

PROCEDURE STATELINK (M : ModuleId ;
                     VAR UseOrNo,OneUse : BITSET ;
                     VAR Duplicate : BOOLEAN
                    ) : Status ;

END Omphale.
```

Annexe E
Implantation des primitives de l'agence Liens

E.1 DuplicateLink

Cette primitive duplique un lien de l'environnement. D'un point de vue interne, la primitive contrôle la validité de la demande et recherche une entrée libre dans l'environnement.

```
Status DuplicateLink (IdLien1,IdLien2)
unsigned int IdLien1,*IdLien2 ;

{
 ContexteTC *ContexteModuleCourant ;
 short int Index1, Index2 ;
 status cr ;
 ModuleId M ;

 ContexteModuleCourant = (ContexteTC *) tache_courante ()  ;
 if (! GoodLink (IdLien1,ContexteModuleCourant,&Index1))
  return (BADIDLINK) ;
  else
     {
      Env *Penv ;
      Lien *Ptl1 , *Ptl2 ;

      Penv = ContexteModuleCourant->Environnement ;
      Ptl1 = & (*Penv) [Index1].LeLien ;
      switch ((*Penv) [Index1].EtatEntree)
          {
           case LienMessage   : return LINKALREADYINBUFFER ;
           case LienADetruire : return LINKDELETED ;
           case LienPresent   :
           case LienPermanent :
                  if (Ptl1->MayDuplicate)
                     {
                      for (Index2 = 0 ;
                            Index2 < SizeOfEnvironnement
                               &&
                            (*Penv) [Index2].EtatEntree != Vide ;
                           Index2++) ;
                      if (Index2 == SizeOfEnvironnement)
                          return NOSPACEINENVIRONNEMENT ;
                          else {
                                Ptl2 = &((*Penv) [Index2].LeLien) ;
                                (*Penv) [Index2].EtatEntree = LienPresent ;
                                M.IdInterne.Version = ++
                                (*Penv) [Index2].Version ;
                                (*Penv) [Index2].LeLien =
                                (*Penv) [Index1].LeLien ;
```

```
                                Ptl2->UseOrNo &= Ptl1->OneOrNUse ;
                                DuplicateNet (Ptl1->SiteName,
                                              Ptl1->Destinataire
                                             ) ;
                                M.IdInterne.Index = Index2 ;
                                *IdLien2 = M.IdExterne ;
                                return GOOD ;
                               }
                        }
                         else return CANNOTDUPLICATETHISLINK ;
                }
        }
}
```

- La fonction *GoodLink* contrôle la validité de l'identificateur du lien à dupliquer. Elle retourne également l'index où se trouve le lien dans l'environnement.
- La duplication n'est possible que si le lien est présent ou permanent. On ne peut dupliquer un lien détruit ou se trouvant dans un message.
- Il faut ensuite vérifier que le lien est duplicable. (Le champ *MayDuplicate* doit être positionné à TRUE). L'erreur CANNOTDUPLICATETHISLINK est retournée si tel n'était pas le cas.
- Recherche d'une entrée vide dans l'environnement. Si cette entrée n'est pas trouvée, l'erreur NOSPACEINENVIRONNEMENT est renvoyée.
- Le compteur de *Version* est incrémenté de 1.
- La copie du lien posséde les mêmes attributs que le lien ayant servi à la copie. Les attributs du lien original ne sont pas modifiés. La copie posséde les mêmes attributs que l'original sauf pour les liens qui n'étaient utilisables qu'une fois. Le programmeur utilisera les primitives *SetNoDups*, *SetNoUse*, *SetOneUse* sur la copie pour modifier les attributs.
- Cette primitive a donc créé une référence supplémentaire vers le module désigné par le lien. Le compteur de référence du module géré par la tâche TN de ce module doit être incrémenté. La primitive *DuplicateNet* émet un message SPART de duplication vers la tâche TN du module pointé par le lien dupliqué. Ce message pourra éventuellement mettre en oeuvre la station de transport.
- Le nouvel identificateur *IdLien2* est retourné a l'application.

E.2 DestroyLink

Cette primitive effectue la destruction d'un lien de l'environnement. La primitive ne fait que noter la demande de destruction. La destruction sera effectuée pendant la phase d'attente du module par la tâche TN.

```
Status DestroyLink (IdLien)
unsigned int IdLien ;
{
 ContexteTC *ContexteModuleCourant ;
 status cr ;
 short int Index ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;
 if (! GoodLink (IdLien,ContexteModuleCourant,&Index))
   return BADIDLINK ;
   else
      switch ((*(ContexteModuleCourant->Environnement)) [Index].EtatEntree)
           {
            case LienPermanent : return CANNOTDELETETHISLINK ;
            case LienMessage : return LINKALREADYINBUFFER ;
            case LienADetruire : return LINKALREADYDELETED ;
            case LienPresent :
                (*(ContexteModuleCourant->Environnement)) [Index].EtatEntree =
                  LienADetruire ;
                return GOOD ;
           }
}
```

- La primitive contrôle la validité de l'identificateur du lien à détruire. (*GoodLink*)
- La destruction d'un lien Permanent est impossible. On ne peut également détruire des liens se trouvant dans des messages.
- Le champ *EtatEntree* est positionné à *LienADetruire*. Nous rappelons que la destruction sera effectuée par la tâche TN pendant la phase de sommeil du module.

E.3 RetrieveLink

Cette primitive permet d'annuler une demande de destruction de lien. La procedure échoue si la primitive *DestroyLink* n'avait pas été utilisée sur le lien pendant la phase d'activité courante.

```
Status RetrieveLink (IdLien)
unsigned int IdLien ;
{
 ContexteTC *ContexteModuleCourant ;
 status cr ;
 short int Index ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;
 if (! GoodLink (IdLien,ContexteModuleCourant,&Index))
    return BADIDLINK ;
    else switch ((*(ContexteModuleCourant->Environnement)) [Index].EtatEntree)
             {
              case LienADetruire :
                 (*(ContexteModuleCourant->Environnement)) [Index].EtatEntree =
                      LienPresent ;
                 return GOOD ;
              default : return LINKWASNOTDELETED ;
             }
}
```

- Validité de l'identificateur de lien *IdLien*. (*GoodLink*)
- Nous vérifions ensuite que la destruction avait bien été demandée. La primitive renvoie *LINKWASNOTDELETED* si tel n'était pas le cas.
- Le champ *EtatEntree* est positionné à *LienPresent*.

E.4 TransferLink

Cette primitive ajoute un lien de l'environnement du module à un message précédemment créé par la primitive *CreateMessage*.

```
Status TransferLink (IdLien,IdMessage)
short int  IdMessage ;
unsigned int IdLien ;
{

 ContexteTC *ContexteModuleCourant ;
 PtMessBuffEmission Ptm ;
 LienInMessage *Ptl ;
 short int Index ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;
 Ptm = SearchMessage (IdMessage,ContexteModuleCourant) ;

 if (!Ptm)
   return BADIDMESSAGE ;
   else if (! GoodLink (IdLien,ContexteModuleCourant,&Index))
          return BADIDLINK ;
          else
             switch ((*(ContexteModuleCourant->Environnement)) [Index].EtatEntree)
                {
                 case LienADetruire : return LINKDELETED ;
                 case LienMessage   : return LINKALREADYINBUFFER ;
                 case LienPermanent : return CANNOTTRANSFERTHISLINK ;
                 case LienPresent :
                      if ((Ptl=(LienInMessage *) NewGlobal
                           (sizeof(LienInMessage)))==NIL)
                        return NOSPACESEGSYSGLOBAL ;
                        else {
                              Ptl->Index = Index ;
                              Ptl->l = ((*ContexteModuleCourant->Environnement))
                              [Index].LeLien ;
                              Ptm->PtMUtil = (PtMessUtilisateur)
                                 AddAbsol (Ptm->PtMUtil) ;
                              Ptl->Suivant = Ptm->PtMUtil->FirstLien ;
                              Ptm->PtMUtil->FirstLien = (LienInMessage *)
                                AddRelat (Ptl) ;
                               Ptm->PtMUtil = (PtMessUtilisateur)
                                 AddRelat (Ptm->PtMUtil) ;
                              (*(ContexteModuleCourant->Environnement))
                                 [Index].EtatEntree = LienMessage ;
                              return GOOD ;
                             }
                }

}
```

- La primitive contrôle la validité du message IdMessage. (fonction *SearchMessage*). Cette fonction a comme paramètre un identificateur de message OMPHALE et renvoie un pointeur sur le message dans la zone d'émission.

```
PtMessBuffEmission SearchMessage (IdMessage,Cont)
  short int IdMessage ;
  ContexteTC *Cont ;
{
  PtMessBuffEmission Pt ;

  if ((IdMessage < 0) || (IdMessage >= Cont->NextNameMessage))
    return NIL ;
    else {
          for (Pt = (PtMessBuffEmission) AddAbsol(Cont->MessAtt->Debut) ;
                (Pt - NIL) && (Pt->Name < IdMessage);
               Pt = (PtMessBuffEmission) AddAbsol (Pt->Suivant)
              ) ;
          if ((Pt == NIL) || (Pt->Name - IdMessage))
            return NIL ;
            else return Pt  ;
         }
}
```

Cette fonction parcourt la liste des messages de la zone d'émission. Elle renvoie *NIL* si le message n'a pas été trouvé ou un pointeur sur le message dans le cas contraire.

- Contrôle de la validité de l'identificateur de lien.(*GoodLink*)
- Si le lien se trouve déjà dans un message (*EtatEntree = LienMessage*), son transfert dans un nouveau message est impossible. Le code d'erreur *LINKALREADYINBUFFER* est alors retourné au programmeur.
- Il n'est pas possible de transférer un lien Permanent dans un message. L'erreur *CANNOTTRANSFERTHISLINK* est alors renvoyée.
- Lorsque tous ces contrôles sont passés correctement, le lien peut être ajouté aux liens accompagnant le message. Les actions suivantes sont alors exécutées:
  - Allocation mémoire nécessaire pour le stockage du lien dans le message.
  - Chainage du lien à la liste des liens du message.
  - L'entrée de l'environnement passe à l'état *LienMessage*.

E.5 TakeAgainLink

Cette primitive supprime un lien qui avait été préalablement placé dans un message.

```
Status TakeAgainLink (IdMessage,IdLien)
short int IdMessage ;
unsigned int IdLien ;
{
 ContexteTC *ContexteModuleCourant ;
 PtMessBuffEmission Pt ;
 LienInMessage *Ptl,*Predl ;
 short int Index ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;
 Pt = SearchMessage (IdMessage,ContexteModuleCourant) ;

 if (!Pt)
   return BADIDMESSAGE ;
   else if (! GoodLink (IdLien,ContexteModuleCourant,&Index))
          return LINKNOTINMESSAGE ;
          else {
                for (Predl=NIL,
                        Ptl = (LienInMessage *) AddAbsol (Pt->PtMUtil->FirstLien);
                        (Ptl - NIL) && (Ptl->Index - Index) ;
                     Predl=Ptl,Ptl = (LienInMessage *) AddAbsol (Ptl->Suivant)
                    ) ;
                if (!Ptl)
                   return LINKNOTINMESSAGE ;
                   else {
                         if (Predl==NIL)
                           Pt->PtMUtil->FirstLien = Ptl->Suivant ;
                           else Predl->Suivant = Ptl->Suivant ;
                         ((*ContexteModuleCourant->Environnement))
                            [Index].EtatEntree = LienPresent ;
                         DisposeGlobal (Ptl,sizeof(LienInMessage)) ;
                         return GOOD ;
                        }
              }
}
```

- La primitive commence par vérifier l'identificateur de message *IdMessage*. La fonction *SearchMessage* recherche le message dans la zone d'émission. Son fonctionnement a été décrit dans le paragraphe précédent.
- Contrôle de la validité de l'identificateur *IdLien*. On utilise comme toujours la fonction *GoodLink*.
- Recherche du lien dans la liste des liens du message. Si il n'y est pas trouvé, l'erreur LINK-NOTINMESSAGE est retournée.

- Destruction du lien de la liste des liens du message. L'espace mémoire utilisé par le lien dans le message est restitué. Le lien redevient alors utilisable. Le champ *EtatEntree* est repositionné à *LienPresent*.

E.6 SetNoUse

Cette primitive permet de restreindre l'ensemble des services accessibles pour un lien. D'un point de vue interne, elle modifie l'attribut *UseOrNo* du lien.

```
Status SetNoUse (IdLien,Services)
unsigned int IdLien ;
unsigned short int Services ;
{
 ContexteTC *ContexteModuleCourant ;
 status cr ;
 short int Index ;

 ContexteModuleCourant = (ContexteTC *) tache_courante ()  ;
 if (! GoodLink (IdLien,ContexteModuleCourant,&Index))
    return (BADIDLINK) ;
    else
       switch ((*(ContexteModuleCourant->Environnement)) [Index].EtatEntree)
             {
              case LienMessage   : return LINKALREADYINBUFFER ;
              case LienADetruire : return LINKDELETED ;
              case LienPermanent : return CANNOTMODIFYTHISLINK ;
              case LienPresent   :
                    ((*ContexteModuleCourant->Environnement)) [Index].LeLien.UseOrNo
                         &= ~Services ;
                    return GOOD ;
             }
}
```

- Validité de l'identificateur de lien *IdLien*.
- Il n'est pas possible de modifier les attributs d'un lien Permanent.
- Le champ *UseOrNo* est recalculé en faisant un et logique avec le complément à 2 des *Services* à invalider.

E.7 SetOneUse

Cette primitive restreint le nombre d'utilisation d'un service. Un service utilisable est une ou une infinité de fois. Le champ *OneOrNUse* indique ce nombre.

```
Status SetOneUse (IdLien,Services)
unsigned int IdLien ;
unsigned short int Services ;
{
 ContexteTC *ContexteModuleCourant ;
 status cr ;
 short int Index ;

 ContexteModuleCourant = (ContexteTC *) tache_courante ()  ;
 if (! GoodLink (IdLien,ContexteModuleCourant,&Index))
   return BADIDLINK ;
   else switch ((*(ContexteModuleCourant->Environnement)) [Index].EtatEntree)
          {
           case LienMessage : return LINKALREADYINBUFFER ;
           case LienADetruire : return LINKDELETED ;
           case LienPermanent : return CANNOTMODIFYTHISLINK ;
           case LienPresent :
                ((*ContexteModuleCourant->Environnement)) [Index].LeLien.OneOrNUse
                   &= ~Services ;
               return GOOD ;
          }
}
```

- Validité de l'identificateur de lien *IdLien*
- Le champ OneOrNUse est recalculé à partir de l'ancienne valeur et des services dont le service ne doit être accessible qu'une seule fois.

E.8 SetNoDups

Cette primitive permet d'interdire la duplication d'un lien.

```
Status SetNoDups (IdLien)
unsigned int IdLien ;
{
 ContexteTC *ContexteModuleCourant ;
 short int Index ;

 ContexteModuleCourant = (ContexteTC *) tache_courante ()  ;

 if (! GoodLink (IdLien,ContexteModuleCourant,&Index))
   return BADIDLINK ;
   else switch ((*(ContexteModuleCourant->Environnement)) [Index].EtatEntree)
           {
            case LienMessage   : return LINKALREADYINBUFFER ;
            case LienADetruire : return LINKDELETED ;
            case LienPermanent : return CANNOTMODIFYTHISLINK ;
            case LienPresent   :
                 ((*ContexteModuleCourant->Environnement)) [Index].LeLien.MayDuplicate
                    = FALSE ;
                 return GOOD ;
           }
}
```

- Vérification de la validité du lien *IdLien*.
- Le champ *MayDuplicate* est positionné à FALSE.

E.9 StateLink

Cette primitive renvoie les attributs du lien. C'est particulièrement utile pour obtenir des informations sur les liens reçus dans les messages.

```
Status StateLink (IdLien,
                  UseOrNo,OneUse,
                  Duplicate
                 )
unsigned IdLien ;
unsigned short int *UseOrNo, *OneUse ;
short int *Duplicate ;
{
 ContexteTC *ContexteModuleCourant ;
 status cr ;
 short int Index ;

 ContexteModuleCourant = (ContexteTC *) tache_courante ()  ;
 if (! GoodLink (IdLien,ContexteModuleCourant,&Index))
   return BADIDLINK ;
   else switch ( (*(ContexteModuleCourant->Environnement)) [Index].EtatEntree)
          {
           case LienMessage   : return LINKALREADYINBUFFER ;
           case LienADetruire : return LINKDELETED ;
           case LienPresent   :
           case LienPermanent :
                {
                 Lien *Ptl ;

                 Ptl = & ((*ContexteModuleCourant->Environnement))
                           [Index].LeLien ;
                 *UseOrNo = Ptl->UseOrNo ;
                 *OneUse = Ptl->OneOrNUse ;
                 *Duplicate = Ptl->MayDuplicate ;
                 return GOOD ;
                }
          }
}
```

- Contrôle de la validité du lien *IdLien*.
- La primitive affecte aux trois paramètres les attributs du lien: *UseOrNo*, *OneOrNUse* et *MayDuplicate*.

E.10 NumberLinkReceived

Cette primitive retourne le nombre de liens reçus dans le message ayant servi au réveil du module.

```
Status NumberLinkReceived (n)
int *n ;
{
 ContexteTC *ContexteModuleCourant ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;
 *n = ContexteModuleCourant->MessAtt->NombreLiensRecus ;
 return GOOD ;
}
```

- La primitive récupère la valeur du champ *NombreLiensRecus* qui se trouve dans la zone de communication *MessAtt*. Cette valeur a été positionnée par la tâche TN au moment du réveil de TC.

E.11 IdLinkReceived

Pour un lien d'un message, la primitive *IdLinkReceived* indique l'endroit où été rangé le lien dans l'environnement.

```
Status IdLinkReceived (i,IdLien)
 int i ;
 short int *IdLien ;
{
 ContexteTC *ContexteModuleCourant ;
 PtMessageAttendre PMA ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;
 PMA = ContexteModuleCourant->MessAtt ;

 if (i > PMA->NombreLiensRecus || i <= 0)
    return ERRORNUMLINK ;
    else {
          *IdLien = PMA->LesLiensRecus [i] ;
          return GOOD ;
         }
}
```

- La primitive renvoie pour le i^{eme} lien du message, sa position dans l'environnement. Les liens du messages ont été placés dans l'environnement par la tâche TN. Les valeurs du

tableau *LiensRecus* ont été initialisées par cette même tâche TN.

Annexe F
Implantation des primitives de l'agence Messages

F.1 CreateMessage

Pour préparer un message avant son envoi, le module doit utiliser cette primitive. Le message est placé dans le tampon (buffer) des messages (zone d'émission). Les actions suivantes sont exécutées :

```
Status CreateMessage (IdLien,IdQuai,IdMessage,l,mess)
unsigned int IdLien ;
short int IdQuai,*IdMessage ;
char *mess ;
short int l ;
{
 PtMessBuffEmission Pt, P ;
 ContexteTC *ContexteModuleCourant ;
 short int Index ;

 ContexteModuleCourant = (ContexteTC *)tache_courante () ;
 if (GoodLink (IdLien,ContexteModuleCourant,&Index))
    switch ( (*(ContexteModuleCourant->Environnement)) [Index].EtatEntree )
        {
         case LienADetruire : return LINKDELETED ;
         case LienMessage   : return LINKALREADYINBUFFER ;
         case LienPresent   :
         case LienPermanent :
                    if (SetOneBit (IdQuai)
                         &
                         (*(ContexteModuleCourant->Environnement))
                          [Index].LeLien.UseOrNo
                         )
                         if ((Pt= (PtMessBuffEmission)
                            NewGlobal (sizeof(MessBuffEmission)))
                          != NIL)
                           {
                             Pt->Suivant = NIL ;
                             Pt->PtMUtil = (PtMessUtilisateur)
                                 NewGlobal (sizeof(MessUtilisateur)) ;
                             if (l)
                               {
                                Pt->PtMUtil->Utile = (char *)
                                                     NewGlobal (l) ;
                                Pt->PtMUtil->Longueur = l ;
                                memcpy (Pt->PtMUtil->Utile,mess,l) ;
                                Pt->PtMUtil->Utile = (char *)
                                           AddRelat (Pt->PtMUtil->Utile) ;
                               }
                              else
                                {
```

```
                                Pt->PtMUtil->Utile = NIL ;
                                Pt->PtMUtil->Longueur = 0 ;
                               } ;

                             Pt->Name = *IdMessage =
                             (ContexteModuleCourant -> NextNameMessage)++ ;
                             Pt->Index = Index ;
                             Pt->PtMUtil->TypeMessage = CodeExterne ;
                             Pt->PtMUtil->Site =
                                (*(ContexteModuleCourant->Environnement))
                                [Index].LeLien.SiteName ;
                             Pt->PtMUtil->Module =
                               (*(ContexteModuleCourant->Environnement))
                               [Index].LeLien.Destinataire ;
                             Pt->PtMUtil->IdQuai = IdQuai ;
                             Pt->PtMUtil->FirstLien = NIL ;
                             Pt->PtMUtil = (PtMessUtilisateur)
                                     AddRelat (Pt->PtMUtil) ;
                             Pt = (PtMessBuffEmission) AddRelat (Pt) ;
                             if (ContexteModuleCourant->MessAtt->Debut == NIL)
                                ContexteModuleCourant->MessAtt->Debut = Pt ;
                                else
                                   {
                                    P = (PtMessBuffEmission) AddAbsol
                                        (ContexteModuleCourant->MessAtt->Fin) ;
                                    P -> Suivant = Pt ;
                                   } ;
                              ContexteModuleCourant->MessAtt->Fin = Pt ;
                              if ((SetOneBit (IdQuai)
                                     &
                                  (*(ContexteModuleCourant->Environnement))
                                      [Index].LeLien.OneOrNUse
                                      ) == 0
                                 )
                                  (*(ContexteModuleCourant->Environnement))
                                       [Index].LeLien.UseOrNo &=
                                         ~(SetOneBit (IdQuai)) ;
                               return GOOD ;
                           }
                           else return NOSPACESEGSYSGLOBAL ;
                else return NOMOREACCESS ;
        } /* switch */
}
```

- La première phase de la primitive consiste à contrôler la validité de l'envoi. La fonction *SetOneBit* positionne le i^{eme} bit d'un mot. On peut ainsi vérifier que le *IdQuai*^{eme} bit du champ *UseOrNo* est positionné à 1 (Quai utilisable).

  – Validité de l'identificateur de lien. (fonction *GoodLink*)

  – On ne peut émettre un message avec un lien dont la destruction a été demandée ou alors avec un lien dont le transfert dans un message a été effectué. Le champ *EtatEntree* permet d'effectuer ce contrôle.

- Le quai *IdQuai* est-il accessible par ce lien? La vérification se fait en utilisant le champ *UseOrNo* du lien.

- Les étapes suivantes sont exécutées que si la phase de contrôle s'est correctement passée.
- Allocation mémoire pour stocker le message.
- Chaînage du message dans la ZE (liste). Le message est ajouté à la fin de la zone d'emission.
- Si le service n'était utilisable qu'une seule fois avec ce lien, le service devient donc inutilisable avec ce lien. On positionne alors le *IdQuai*^{eme} bit de *UseOrNo* à 0.
- Retourne l'identificateur de message.

F.2 DeleteMessage

Cette primitive détruit un message de la zone d'émission. Si le message contenait des liens, ces liens sont automatiquement replacés dans l'environnement. L'espace mémoire réservé pour le message est restitué. Les droits sur le lien utilisé sont remis à leur état initial.

```
Status DeleteMessage (IdMessage)
short int IdMessage ;
{
 ContexteTC *ContexteModuleCourant ;
 MessBuffEmission *Pt,*PtPred ;

 ContexteModuleCourant = (ContexteTC *)tache_courante () ;
 if ((IdMessage < 0) || (IdMessage >= ContexteModuleCourant-> NextNameMessage))
   return BADIDMESSAGE ;
   else
      {
       for (PtPred=NIL,
              Pt = (MessBuffEmission *)
                  AddAbsol (ContexteModuleCourant->MessAtt->Debut) ;
            (Pt - NIL) && (Pt->Name < IdMessage) ;
            PtPred = Pt,Pt = (MessBuffEmission *) AddAbsol (Pt->Suivant)
           ) ;
       if ((Pt == NIL) || (Pt->Name - IdMessage))
          return BADIDMESSAGE ;
          else
             {
              PtMessUtilisateur pm ;

              pm = (PtMessUtilisateur) AddAbsol (Pt->PtMUtil) ;
              if (pm->Longueur)
                 DisposeGlobal (AddAbsol (pm->Utile),pm->Longueur) ;
              ReprendreLiens (ContexteModuleCourant,pm->FirstLien) ;

              if ((SetOneBit (pm->IdQuai)
                  &
                  (*(ContexteModuleCourant->Environnement))
                 [Pt->Index].LeLien.UseOrNo
                   ) == 0
                 )
                  (*(ContexteModuleCourant->Environnement))
                      [Pt->Index].LeLien.UseOrNo |=
                          SetOneBit (pm->IdQuai) ;

               if (PtPred == NIL)
                   {
                    ContexteModuleCourant->MessAtt->Debut= Pt->Suivant ;
                    if (Pt->Suivant == NIL)
                        ContexteModuleCourant->MessAtt->Fin=NIL ;
                   }
                 else PtPred->Suivant=Pt->Suivant ;
```

```
            DisposeGlobal (pm,sizeof (MessUtilisateur)) ;
            DisposeGlobal (Pt,sizeof(MessBuffEmission)) ;
            return GOOD ;
         }
      }
}
```

- Le paramètre *IdMessage* désigne l'identificateur du message à détruire.
- La zone d'émission est parcourue pour retrouver le message. (Le pointeur *Pt*). Si ce pointeur se retrouve à *NIL*, c'est que le message n'existait pas et l'erreur *BADIDMESSAGE* est renvoyée.
- L'espace mémoire réservé pour le message est restitué.
- Les liens du message sont remis dans l'environnement. (fonction *ReprendreLiens*).

```
void ReprendreLiens (Cont,P)
LienInMessage *P ;
ContexteTC *Cont ;
{
 LienInMessage *pc,*pp ;

 for (pc = (LienInMessage *) AddAbsol (P);(pc - NIL);)
    {
     (*(Cont->Environnement)) [pc->Index].EtatEntree = LienPresent ;

     pp = pc ;
     pc = (LienInMessage *) AddAbsol (pc->Suivant) ;
     DisposeGlobal (pp,sizeof(LienInMessage)) ;
    }
}
```

La liste des liens du message est parcourue. Pour chaque lien, le lien devient réutilisable. (Le champ *EtatEntree* est repositionné à *LienPresent*). L'espace mémoire utilisé par chaque lien du message est libéré.

- Le message est retiré de la zone d'émission. Le chaînage des messages est mis à jour.
- Si le quai n'est pas accessible avec le lien ayant servi à l'émission, le quai n'était accessible qu'une seule fois. Le programmeur cherche à annuler cette utilisation. Il faut donc rendre le lien réutilisable. Le champ *UseOrNo* est correctement recalculé.
- L'espace mémoire pour le stockage du message est restitué.

F.3 DeleteAllMessages

Cette primitive détruit l'ensemble des messages se trouvant dans la zone d'émission.

```
Status DeleteAllMessages ()
{
 ContexteTC *ContexteModuleCourant ;
 MessBuffEmission *pc,*pp ;
 PtMessUtilisateur Ptu ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;

 for (pc = (MessBuffEmission *) AddAbsol (ContexteModuleCourant->MessAtt->Debut) ;
       (pc - NIL) ;)
    {
     Ptu = (PtMessUtilisateur) AddAbsol (pc->PtMUtil) ;
     if (Ptu->Longueur)
        DisposeGlobal (AddAbsol (Ptu->Utile),Ptu->Longueur) ;
     ReprendreLiens (ContexteModuleCourant,Ptu->FirstLien) ;

     if ((SetOneBit (Ptu->IdQuai)
            &
          (*(ContexteModuleCourant->Environnement)) [pc->Index].LeLien.UseOrNo
          ) == 0
         )
         (*(ContexteModuleCourant->Environnement)) [pc->Index].LeLien.UseOrNo |=
                  SetOneBit (Ptu->IdQuai) ;

     DisposeGlobal (Ptu,sizeof (MessUtilisateur)) ;

     pp = pc ;
     pc = (MessBuffEmission *) AddAbsol (pc->Suivant) ;
     DisposeGlobal (pp,sizeof (MessBuffEmission)) ;
    } ;
 ContexteModuleCourant->NextNameMessage = 0 ;
 return GOOD ;
}
```

- Parcours et destruction de la liste des messages de la zone d'émission.
- Pour chaque message, il faut détruire la zone mémoire allouée pour le message.
- Il faut également détruire l'ensemble des liens accompagnant le message (fonction *ReprendreLiens*).
- Le champ *NextNameMessage* est remis à zéro. Ce champ désigne le prochain identificateur de message qui sera renvoyé par la primitive *CreateMessage*.

F.4 SetServ

Cette primitive précise les quais (services) ouverts et donc succeptibles de conduire à une réactivation sur un de ces quais.

```
Status SetServ (Services)
unsigned short int Services ;
{
 Contexte *ContexteModuleCourant ;

 ContexteModuleCourant = (Contexte *) tache_courante () ;
 ContexteModuleCourant->MessAtt->QuaisAOuvrir = Services ;
 return GOOD ;
}
```

- La primitive affecte son paramètre au champ *QuaisAOuvrir* de la zone de communication *MessAtt*.

F.5 PutServ

La primitive *PutServ* ajoute aux quais déjà ouverts une liste de quais.

```
Status PutServ (Services)
unsigned short int Services ;
{
 Contexte *ContexteModuleCourant ;

 ContexteModuleCourant = (Contexte *) tache_courante () ;
 ContexteModuleCourant->MessAtt->QuaisAOuvrir |= Services ;
 return GOOD ;
}
```

- Un ou logique est effectué avec les *Services* passés en paramétres.

F.6 ResServ

Cette primitive invalide des quais ouverts. Elle ferme des quais.

```
Status ResServ (Services)
unsigned short int Services ;
{
 Contexte *ContexteModuleCourant ;

 ContexteModuleCourant = (Contexte *) tache_courante () ;
 ContexteModuleCourant->MessAtt->QuaisAOuvrir &= ~Services ;
 return GOOD ;
}
```

F.7 ReadServ

Cette primitive renvoie la valeur de *QuaisAOuvrir* de la zone de communication.

```
Status ReadServ (Services)
 unsigned short int *Services ;
{
 Contexte *ContexteModuleCourant ;

 ContexteModuleCourant = (Contexte *) tache_courante () ;
 *Services = ContexteModuleCourant->MessAtt->QuaisAOuvrir ;
 return GOOD ;
}
```

F.8 SizeMessage

Cette primitive fournit la longueur du message reçu pour la phase d'activité courante.

```
Status SizeMessage ()
{
 ContexteTC *ContexteModuleCourant ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;
 return ContexteModuleCourant->MessAtt->Longueur ;
}
```

F.9 LastMessage

La primitive *LastMessage* renvoie la valeur du dernier message reçu. Elle retourne la partie utile du message sans les liens. Le programmeur doit s'assurer que la zone mémoire qu'il fournit à la primitive *LastMessage* est de taille suffisante. Il peut connaitre la taille de cette zone en utilisant préalablement la primitive *SizeMessage*.

```
Status LastMessage (ch)
char *ch ;
{
 ContexteTC *ContexteModuleCourant ;
 PtMessageAttendre PMA ;
 char *p ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;
 PMA = ContexteModuleCourant->MessAtt ;
 p = (char *) AddAbsol (PMA->MessageRecu) ;
 memcpy (ch,p,PMA->Longueur) ;
 return GOOD ;
}
```

- Seul le message reçu au cours de la phase d'activité courante est accessible.

F.10 Wait

La primitive Wait marque la fin de la phase d'activité du module. La tâche TC exécute donc la primitive *Wait* de l'agence Message.

```
Status Wait ()
{
 status Service ;
 ContexteTC *ContexteModuleCourant ;
 char MessSpart [MESS_LENGTH] ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;

 MessSpart [0] = CodeAttendre ;
 ContexteModuleCourant->MessAtt->Normal = TRUE ;
 send (MessSpart,ContexteModuleCourant->MbxIdTN,0) ;

 receive  (MessSpart,ContexteModuleCourant->MbxIdTC,INFINITE) ;

 if (MessSpart [0] == CodeDetruire)
     FinTacheTC (ContexteModuleCourant) ;

 ContexteModuleCourant->NextNameMessage = 0 ;
 memcpy (&Service,MessSpart+1,sizeof(status)) ;
 return Service ;
}
```

Pendant la phase d'activité, le module a déposé (primitive *CreateMessage*) dans sa ZE des messages OMPHALE en vue d'un transfert. L'appel à la primitive Wait marque la fin de la phase d'activité du module.

- TC émet un message SPART vers TN lui précisant que le module se met en attente (TC est en attente). TC attend une réponse sur la boîte aux lettres MbxIdTC.
- TN récupére le message émis par TC. Les actions exécutées par TN seront étudiées plus précisement dans la section routage du présent chapitre.
  - TN est chargée de l'émission des messages de la ZE. Les messages distants sont passés au système de transport qui assure le transfert des messages. Les messages locaux sont directement envoyés au module (la tâche TN) destinataire.
  - Dans certaines situations, TN décide de la fin du module. Si tous les quais sont fermés, le module n'est plus reactivable. De plus si le module n'est plus référencé dans le système et que les quais ouverts sont vides, le module ne sera jamais réactivé. Un message de destruction est alors envoyé à la tâche TC.
  - TN choisit un service et un message sur un des quais ouverts. Un message SPART de réactivation est alors émis à destination de la tâche TC.

- TC reçoit le message SPART de TN. Deux possibilités:
  - Un message de Destruction. C'est la fin du module et la tâche TC doit donc s'arrêter. La fonction FinTacheTC stoppe l'exécution de TC et assure la destruction de la boîte aux lettres *MbxIdTC*.

```
void FinTacheTC (Cont)
  ContexteTC *Cont ;
{
 xx_id MbIdRelai ;
 MessGT *PtName ;
 char MessSpartRelai [MESS_LENGTH] ;
 general_address ga ;
 task_obj *task = NIL ;

 find (GLOBAL,"BLREL",&MbIdRelai) ;
 alloc_memory (Cont->SegIdGT,sizeof(MessGT),&PtName) ;
 PtName = (MessGT *) (Cont->BaseGT + (unsigned int) PtName) ;
 PtName->Type = Delete ;
 task = (task_obj *) access_obj (Cont->ctx_sys.task_id) ;
 PtName->mbn = task->prog_id ;
 gen_addr (PtName,&ga) ;
 memcpy (MessSpartRelai,&ga,sizeof(general_address)) ;
 send (MessSpartRelai,MbIdRelai,0) ;

 delete_mbx (Cont->MbxIdTC) ;

 exit_task () ;
}
```

    - La tâche *Relai* est responsable de la gestion des tâches calcul TC. Elle assure la création de l'ensemble des tâches calculs du site. Son fonctionnement sera plus précisément décrit dans la section Création et destruction de module. La tâche *Relai* est prévenue par la tâche TC. L'image binaire de la tâche pourra éventuellement être déchargée.
    - La boîte aux lettres *MbxIdTC* est détruite.
    - La tâche TC s'arrête (*exit_task*).
  - Un message de continuation. Le module est actif et le message reçu contient le numéro du service choisi par TN. La zone de communication MessAtt contient le message reçu ainsi que les liens accompagnant le message.

F.11 Failure

Comme tout être humain, le programmeur est sujet à erreurs et défaillance. Les problèmes doivent être récupérés par le noyau. La primitive Failure est exécutée en cas de problème. Le traitement est analogue au *Wait* sauf que le champ *Normal* est positionné à FALSE. Les messages de la zone d'émission seront détruits par la tâche TN.

```
void Failure (ect)
 ex_ctx_type *ect ;
{
 ContexteTC *ContexteModuleCourant ;
 char MessSpart [MESS_LENGTH] ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;
 ContexteModuleCourant->MessAtt->Normal = FALSE ;
 MessSpart [0] = CodeAttendre ;
 send (MessSpart,ContexteModuleCourant->MbxIdTN,0) ;

 receive  (MessSpart,ContexteModuleCourant->MbxIdTC,INFINITE) ;

 FinTacheTC (ContexteModuleCourant) ;
}
```

- La champ *Normal* est positionné à FALSE.
- La tâche TC envoie un message d'attente à TN.
- La fonction *FinTacheTC* détruit la boîte aux lettres MbxIdTC et la tâche TC.
- La primitive *Failure* fait donc office de traitement d'exception (handler). La connexion se fait par la primitive SPART *ex_connect*:

  ```
  ex_connect (Failure)
  ```

  Cette connexion doit être effectuée au début de l'exécution de la tâche TC après la primitive *InitBuffer*.
  En cas d'exception provoquée par la tâche TC, il y a déroutement et la procédure *Failure* est exécutée. La procédure d'exception a accès à toutes les primitives de toutes les agences.

F.12 InitBuffer

Cette fonction initialise les données du contexte du module. *InitBuffer* initialise les données du contexte de la tâche TC. Les données du contexte partagées avec TN sont initialisées par TN. Les données du contexte locales à TN sont initialisées par TN. La primitive *InitBuffer* doit être appelée au début de l'exécution du module.

```
Status InitBuffer ()
{
 ContexteTC *ContexteModuleCourant ;
 char MessInit [MESS_LENGTH] ;
 unsigned int P ;
 task_obj *task = NIL ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;
 receive (Mess,0,0) ;
 memcpy (&ContexteModuleCourant->MbxIdTN,MessInit,sizeof(xx_id)) ;
 memcpy (&ContexteModuleCourant->MbxIdTC,MessInit+sizeof(xx_id),sizeof(xx_id));
 ContexteModuleCourant->NextNameMessage = 0 ;

 find (GLOBAL,"000001",&ContexteModuleCourant->SegIdSyst) ;
 access_segment (ContexteModuleCourant->SegIdSyst,
                 READ_WRITE,&ContexteModuleCourant->BaseSyst) ;

 find (GLOBAL,"BLGT",&ContexteModuleCourant->MbIdGT) ;

 find (GLOBAL,"SEGT",&ContexteModuleCourant->SegIdGT) ;
 access_segment (ContexteModuleCourant->SegIdGT,READ_WRITE,
                 &ContexteModuleCourant->BaseGT) ;

 receive (MessInit,ContexteModuleCourant->MbxIdTC,INFINITE) ;

 memcpy (&P,MessInit,sizeof (unsigned int)) ;
 ContexteModuleCourant->Environnement = (Env *) AddAbsol (P) ;

 memcpy (&P,MessInit+sizeof (unsigned int),sizeof (unsigned int)) ;
 ContexteModuleCourant->MessAtt = (PtMessageAttendre) AddAbsol (P) ;

 return GOOD ;
}
```

- La tâche TC doit connaître les identificateurs des boîtes aux lettres *MbxIdTN* et *MbxIdTC*. Ces identificateurs sont passés en paramètres à la tâche TC. Le paramètre initial d'une tâche est recupéré par la primitive de lecture de boîte aux lettres avec l'identificateur de boîte aux lettres nul.
- La tâche TC doit accéder au segment système *SegSyst*.

- *SegGT* est un segment mémoire nécessaire à la communication avec la tâche GT qui est responsable de la création de module sur le site.
- TC récupère ensuite le pointeur sur l'environnement et sur la zone de communication *MessAtt*. Ces zones ont été allouées et initialisées par TN. Ces pointeurs sont reçus dans un message sur la boîte aux lettres *MbxIdTC*.

F.13 CreateModule

Cette primitive du noyau est exécutée par les modules gestionnaire de module. Elle a comme paramétre un nom de module et renvoie l'identificateur du lien vers le module créé.

```
Status CreateModule (FileName,IdLien)
 char FileName [] ;
unsigned int *IdLien ;
{
 ContexteTC *ContexteModuleCourant ;
 MessGT *PtName ;
 general_address ga ;
 char MessSpart [MESS_LENGTH] ;
 Lien LienModule ;
 int i ;
 ModuleId M ;

 ContexteModuleCourant = (ContexteTC *) tache_courante () ;
 alloc_memory (ContexteModuleCourant->SegIdGT,sizeof(MessGT),&PtName) ;
 PtName = (MessGT *) (ContexteModuleCourant->BaseGT + (unsigned int) PtName) ;
 PtName->Type = Create ;
 PtName->ModuleDemandeur = ContexteModuleCourant->MbxIdTC ;
 strcpy (PtName->proto,FileName) ;
 gen_addr (PtName,&ga) ;
 memcpy (MessSpart,&ga,sizeof(general_address)) ;
 send (MessSpart,ContexteModuleCourant->MbIdGT,0) ;
 receive (MessSpart,ContexteModuleCourant->MbxIdTC,INFINITE) ;

 if (MessSpart [0] == OK)
    {
     memcpy (&LienModule.Destinataire,MessSpart+1,sizeof(xx_id)) ;
     LienModule.SiteName = MyHouse ;
     LienModule.UseOrNo =  -1 ;
     LienModule.OneOrNUse = -1 ;
     LienModule.MayDuplicate = TRUE ;
     for (i=0;
          i<SizeOfEnvironnement &&
          ((*ContexteModuleCourant->Environnement)) [i].EtatEntree - Vide;
          i++) ;
     if (i == SizeOfEnvironnement)
        return NOSPACEINENVIRONNEMENT ;
        else
           {
            ((*ContexteModuleCourant->Environnement)) [i].EtatEntree =
                LienPresent ;
            M.IdInterne.Version = ++
            ((*ContexteModuleCourant->Environnement)) [i].Version ;
            ((*ContexteModuleCourant->Environnement)) [i].LeLien = LienModule ;
            M.IdInterne.Index = i ;
            *IdLien = M.IdExterne ;
            return GOOD ;
           }
```

```
    }
     else return ERRORCREATEMODULE ;
}
```

- La primitive *CreateModule* a deux paramètres qui sont:
  - *FileName*: Nom du module (fichier) contenant le code de la tâche TC.
  - *IdLien*: Identificateur du lien vers le module crée.

  La requête sera traitée par l'une des tâches *GT* du site. Un message est donc émis et les paramètres sont recopiés dans le segment *SegIdGT*.
  - Allocation mémoire dans le segment *SegIdGT*. Les paramètres de la requête y sont recopiés dans ce segment.
  - Envoi d'un message dans la boîte aux lettres *MbIdGT*. Le message sera traité par l'une des tâches *GT* du site.
- La tâche *GT* ayant traité la requête envoie sa réponse au module.
  - Le premier octet contient le status de la requête. Les deux octets suivants contiennent le numéro de la boîte aux lettres MBN du module créé. Ce numéro servira à la création du lien dans l'environnement.
  - Le lien vers le module créé est placé dans l'environnement du module. L'identificateur du lien dans l'environnement est renvoyé par le paramètre *IdLien*.

Sur cette implantation de l'architecture OMPHALE, la lecture de l'image binaire du module, l'allocation mémoire (segments de Code, Pile) sont internes au noyau. Ces fonctions sont normalement à la charge de modules système du CORTEX. Certains mécanismes du noyau SPART ne peuvent être contournés.

Annexe G
Implantation des primitives de l'agence Routage

Les primitives décrites dans cette section sont exécutées par la tâche TN. Le code de la tâche TN est étudié dans la section suivante et montre comment sont appelées les primitives de l'agence Routage.

G.1 Dupliquer

La tâche TN a reçu un message SPART lui indiquant qu'une duplication de lien avait été effectuée. Le traitement à exécuter consiste donc à incrémenter la valeur du compteur de duplication (*DuplicateCount*).

```
status Dupliquer ()
{
 ContexteTN *Cont ;
 ContexteOmphale PCO ;

 Cont = (ContexteTN *) tache_courante () ;
 PCO = Cont->CtxSysOmphale ;

 PCO->DuplicateCount ++ ;
 return GOOD ;
}
```

G.2 Detruire

La tâche TN a reçu un message de destruction de référence. Le compteur de référence *DuplicateCount* doit donc être décrémenté.

```
status Detruire ()
{
 ContexteTN *Cont ;
 ContexteOmphale PCO ;

 Cont = (ContexteTN *) tache_courante () ;
 PCO = Cont->CtxSysOmphale ;

 PCO->DuplicateCount -- ;
 switch (PCO->Etat)
      {
       case Mourant :
                 if (PCO->DuplicateCount == 0)
                    DetruireTN (PCO) ;
                 return GOOD ;

       case EnAttente :
                  if (PCO->DuplicateCount == NombreAutoReference (PCO))
                      DevientMourant (PCO) ;
                  return GOOD ;
       default : return GOOD ;
      }
}
```

- La valeur du compteur *DuplicateCount* est décrémentée de 1.
- Si le module était **Mourant** et que la valeur de *DuplicateCount* passe à 0, le module peut mourir. La mort du module consiste à détruire la boîte aux lettres *MbxIdTN* et à détruire la tâche TN. L'espace mémoire alloué pour le contexte OMPHALE est restitué (fonction *DisposeLocalTN*). Ces actions sont réalisées par la procédure *DetruireTN*.

```
void DetruireTN (PCO)
ContexteOmphale PCO ;
{
  delete_mbx (PCO->MbxIdTN) ;
  DisposeLocalTN (PCO,sizeof (StructContexteOmphale)) ;
  exit_task () ;
}
```

  Les autres données du contexte avaient été libérées au moment du passage à l'état **Mourant**.
- Si le module était en **Attente** et non référençable, le module peut commencer à mourir. En fait, il est le seul à se connaitre. La fonction *NombreAutoReference* calcule le nombre d'auto-

liens se trouvant dans l'environnement et la zone de réception. Un module n'est plus référençable si *NombreAutoReference = DuplicateCount.*

```
char CestMoi (PCO,i)
ContexteOmphale PCO ;
short int i ;
{
 return ( (PCO->MbxIdTN == (*(PCO->Environnement)) [i].LeLien.Destinataire)
           &&
          (MyHouse == (*(PCO->Environnement)) [i].LeLien.SiteName)
        ) ;
}

short int NombreReferenceMessage (PCO,P)
ContexteOmphale PCO ;
LienInMessage *P ;
{
 short int nb = 0 ;
 LienInMessage *pc ;

 for (pc = (LienInMessage *) AddAbsolTN (P) ;
       pc - NIL ;
      pc = (LienInMessage *) AddAbsolTN (pc->Suivant)
     )
     if ( (pc->l.SiteName == MyHouse)
          &&
          (pc->l.Destinataire == PCO->MbxIdTN)
        )
        nb ++ ;
  return nb ;
}

short int NombreReferenceQuai (PCO,i)
ContexteOmphale PCO ;
short int i ;
{
 MessageSurQuai *pc ;
 short int nb = 0 ;
 PtMessUtilisateur pt ;

 for (pc = PCO->Quai [i].Tete ; pc - NIL ; pc = pc->Suivant)
   {
    pt = pc->PtMUtil ;
    nb += NombreReferenceMessage (PCO,pt->FirstLien) ;
   }
 return nb ;
}

short int AutoReferenceZR (PCO)
ContexteOmphale PCO ;
{
 short int i, nb = 0 ;

 for (i = 0 ; i < MAXSERVICES; i++)
```

```
    nb += NombreReferenceQuai (PCO,i) ;
 return nb ;
}

short int AutoReferenceEnv (PCO)
ContexteOmphale PCO ;
{
 short int i, nb = 0 ;

 for (i = 3 ; i < SizeOfEnvironnement; i++)
   if (
       ((*(PCO->Environnement)) [i].EtatEntree == LienPresent)
          &
       CestMoi (PCO,i)
      )
      nb++ ;
  return 1+nb ;
}

short int NombreAutoReference (PCO)
ContexteOmphale PCO ;
{
 return (AutoReferenceEnv (PCO) + AutoReferenceZR (PCO)) ;
}
```

Calculer le nombre d'auto-liens se fait en calculant le nombre d'auto-liens de la zone de réception (fonction *AutoReferenceZR*) et le nombre d'auto-liens de l'environnement (fonction *AutoReferenceEnv*).

Le passage à l'état **Mourant** entraine l'exécution de la fonction *DevientMourant*. L'étude de cette fonction sera faite ultérieurement.

G.3 Externe

Le module a reçu un message OMPHALE provenant d'un autre module. Cela se traduit par la réception au niveau de la tâche TN d'un message SPART contenant l'adresse du MESSAGE OMPHALE. La tâche TN exécute alors la primitive *Externe* de l'agence Routage.

```
status Externe (MessSpart)
char MessSpart [] ;
{
 MessageSurQuai *Pmsq ;
 PtMessUtilisateur Ptu ;
 ContexteTN *Cont ;
 ContexteOmphale PCO ;

 Cont = (ContexteTN *) tache_courante () ;
 PCO = Cont->CtxSysOmphale ;

 memcpy (&Ptu,MessSpart+1,sizeof (unsigned int)) ;
 Ptu = (PtMessUtilisateur) AddAbsolTN (Ptu) ;
 if (PCO->Etat == Mourant)
   {
    DetruireMessageRecu (Ptu) ;
    return GOOD ;
   }
  else
     {
      Pmsq = (MessageSurQuai *) NewLocalTN (sizeof(MessageSurQuai)) ;
      Pmsq->PtMUtil = Ptu ;
      Pmsq->Suivant = NIL ;
      get_time (&Pmsq->DateArrivee) ;
      if (PCO->Quai [Ptu->IdQuai].Tete == NIL)
        {
         PCO->Quai [Ptu->IdQuai].Tete = PCO->Quai [Ptu->IdQuai].Queue = Pmsq ;
         PCO->QuaisOuIlyaDesMessages |= SetOneBitTN (Ptu->IdQuai) ;
        }
       else {
             PCO->Quai [Ptu->IdQuai].Queue->Suivant = Pmsq ;
             PCO->Quai [Ptu->IdQuai].Queue = Pmsq ;
            } ;
      if ((PCO->Etat == EnAttente)
           &&
             (
              PCO->QuaisOuverts
                &
              PCO->QuaisOuIlyaDesMessages
             )
          )
        ActiverTC (PCO) ;
        return GOOD ;
```

```
    }
}
```

- Si le module est **mourant**, le message est détruit ainsi que les liens l'accompagnant. (fonction *DetruireMessageRecu*).

```
void DetruireMessageRecu (P)
PtMessUtilisateur P ;
{
 if (P->Longueur)
    DisposeGlobalTN (AddAbsolTN (P->Utile),P->Longueur) ;
 DetruireLienMessage (P->FirstLien) ;
 DisposeGlobalTN (P,sizeof(MessUtilisateur)) ;
}
```

Cette fonction libére l'espace mémoire du message. Les liens contenus dans le message sont détruits par la fonction *DetruireLienMessage*.

```
void DetruireLienMessage (P)
LienInMessage *P ;
{
 LienInMessage *pc,*pp ;

 for (pc = (LienInMessage *) AddAbsolTN (P);(pc - NIL);)
   {
    DestroyNet (pc->l.SiteName,pc->l.Destinataire) ;

    pp = pc ;
    pc = (LienInMessage *) AddAbsolTN (pc->Suivant) ;
    DisposeGlobalTN (pp,sizeof(LienInMessage)) ;
   }
}
```

La liste des liens du message est parcourue complétement. Un message de destruction de référence est émis à l'intention de la tâche TN du module pointé par le lien (fonction *DestroyNet*). L'espace mémoire utilisé pour le stockage du lien dans le message est libéré.

- Si le module n'est pas **mourant** (**Actif** ou **Attente**), le message est placé sur le quai. Le message est daté (fonction *get_time*). Le message est chaîné aux autres messages du quai. Si le module est en atente et que l'un des quais ouverts n'est pas vide, le module est reactivé. (fonction *ActiverTC*)

```
void ActiverTC (PCO)
ContexteOmphale PCO ;
{
 status cr,QuaiChoisi ;
 char MessSpart [MESS_LENGTH] ;
 MessageSurQuai *Pmsq ;
 LienInMessage *Ptl ;
```

```
QuaiChoisi = QuaiAvecLeVieuxMessage (PCO) ;

PCO->MessAtt->MessageRecu = PCO->Quai [QuaiChoisi].Tete->PtMUtil->Utile ;
PCO->MessAtt->Longueur = PCO->Quai [QuaiChoisi].Tete->PtMUtil->Longueur ;

PCO->MessAtt->NombreLiensRecus = 0 ;
Ptl = (LienInMessage *)
          AddAbsolTN PCO->Quai [QuaiChoisi].Tete->PtMUtil->FirstLien) ;
while (Ptl)
   {
    PCO->MessAtt->NombreLiensRecus ++ ;
    PCO->MessAtt->LesLiensRecus [PCO->MessAtt->NombreLiensRecus] =
                     Ajouter (PCO,Ptl->l) ;
    Ptl = (LienInMessage *) AddAbsolTN (Ptl->Suivant) ;
   } ;

DisposeGlobalTN (PCO->Quai [QuaiChoisi].Tete->PtMUtil,
                 sizeof(MessUtilisateur)
                ) ;

Pmsq = PCO->Quai [QuaiChoisi].Tete ;
PCO->Quai [QuaiChoisi].Tete = PCO->Quai [QuaiChoisi].Tete->Suivant ;
DisposeLocalTN (Pmsq,sizeof (MessageSurQuai)) ;

if (PCO->Quai [QuaiChoisi].Tete == NIL)
   PCO->QuaisOuIlyaDesMessages &= (~SetOneBitTN (QuaiChoisi)) ;

MessSpart [0] = CodeExecution ;
memcpy (MessSpart+1,&QuaiChoisi,sizeof(status)) ;

cr = send (MessSpart,PCO->MbxIdTC,0) ;
PCO->Etat = EnCours ;
}
```

Les liens du message sont ajoutés (fonction *Ajouter* à l'environnement du module. Les identificateurs de liens sont placés dans le tableau *LesLiensRecus*.
Parmi les files d'attentes ouvertes et qui ne sont pas vides, la fonction *QuaiAvecLeVieuxMessage* détermine le message le plus ancien de la zone de réception. Elle balaye la zone de réception en comparant les dates d'arrivée des différents messages. Elle choisit le quai avec le message le plus ancien (la plus petite date).

```
status QuaiAvecLeVieuxMessage (PCO)
ContexteOmphale PCO ;
{
 status i = 0,QuaiGagnant=0,j ;
 unsigned int HeureDuGagnant = (-1) ;


 for (;i < MAXSERVICES;i++)
    {
     if (PCO->Quai[i].Tete - NIL)
        {
         if ((PCO->Quai[i].Tete->DateArrivee < HeureDuGagnant)
```

```
                &&
               (PCO->QuaisOuverts  SetOneBitTN (i)
              )
               {
                QuaiGagnant = i ;
                HeureDuGagnant = PCO->Quai [i].Tete->DateArrivee ;
               }
          }
    }
  return QuaiGagnant ;
}
```

Les liens accompagnant le message sont ajoutés dans l'environnement. Les identificateurs de liens sont placés dans les tableau *LesLiensRecus*.

L'espace mémoire utilisé par le message dans la zone de réception est libéré.

La tâche TC est réactivée par un message SPART envoyé sur *MbxIdTC*. Le message contient le numéro du service (quai). La structure *MessAtt* contient les références du messages OMPHALE ainsi que les identificateurs des liens se trouvant dans le message.

G.4 Attendre

Lorsque le module se met en attente, TC émet un message SPART à l'intention de la tâche TN. A la réception de ce message, TN exécute la primitive *Attendre* de l'agence Routage.

```
status Attendre ()
 {
  status cr ;
  unsigned int Site, Pemis, i ;
  LienInMessage *Ptl ;
  char MessSpart [MESS_LENGTH] ;
  PtMessBuffEmission Ptm, P ;
  ContexteTN *Cont ;
  ContexteOmphale PCO ;

  Cont = (ContexteTN *) tache_courante () ;
  PCO = Cont->CtxSysOmphale ;

  if (PCO->MessAtt->Longueur)
    {
     PCO->MessAtt->MessageRecu = (char *)
                                 AddAbsolTN (PCO->MessAtt->MessageRecu) ;
     DisposeGlobalTN (PCO->MessAtt->MessageRecu,PCO->MessAtt->Longueur) ;
    } ;

  PCO->MessAtt->Longueur = 0 ;
  PCO->MessAtt->MessageRecu = NIL ;

  if (PCO->MessAtt->Normal)
    {
     PCO->QuaisOuverts = PCO->MessAtt->QuaisAOuvrir ;
     Ptm = (PtMessBuffEmission) AddAbsolTN (PCO->MessAtt->Debut) ;
     PCO->MessAtt->Debut = NIL ;

     MessSpart [0] = CodeExterne ;
     while (Ptm)
          {

           Ptm->PtMUtil = (PtMessUtilisateur) AddAbsolTN (Ptm->PtMUtil) ;
           Ptl = (LienInMessage *) AddAbsolTN (Ptm->PtMUtil->FirstLien) ;

           while (Ptl)
                {
                 (*(PCO->Environnement)) [Ptl->Index].EtatEntree = Vide ;
                 Ptl = (LienInMessage *) AddAbsolTN (Ptl->Suivant) ;
                } ;
           Pemis = AddRelatTN (Ptm->PtMUtil) ;
           memcpy (MessSpart+1,&Pemis,sizeof(unsigned int)) ;
           Site = Ptm->PtMUtil->Site ;

           if (Site == MyHouse)
              send (MessSpart,Ptm->PtMUtil->Module,0) ;
```

```
            else EnvoiST (Ptm->PtMUtil) ;

          P = Ptm ;
          Ptm = (PtMessBuffEmission) AddAbsolTN (Ptm->Suivant) ;
          DisposeGlobalTN (P,sizeof(MessBuffEmission)) ;
         }
      }
 else {
       PCO->QuaisOuverts = 0 ;
       DetruireMessageZE (PCO) ;
      }

for (i=0;i<SizeOfEnvironnement;i++)
   if ( ((*(PCO->Environnement)) [i].EtatEntree == LienADetruire))
     {
      (*(PCO->Environnement)) [i].EtatEntree = Vide ;
      DestroyNet ((*(PCO->Environnement)) [i].LeLien.SiteName,
                  (*(PCO->Environnement)) [i].LeLien.Destinataire) ;
     }

if (PCO->QuaisOuverts & PCO->QuaisOuIlyaDesMessages)
  {
   ActiverTC (PCO) ;
   return GOOD ;
  }
 else if (PCO->QuaisOuverts)
        {if (PCO->DuplicateCount == NombreAutoReference (PCO))
           {
            DevientMourant (PCO) ;
            return GOOD ;
            }
          PCO->Etat = EnAttente ;
          return GOOD ;
        }
      else {
            DevientMourant (PCO) ;
            if (PCO->DuplicateCount == 0)
               DetruireTN (PCO) ;
            return GOOD ;
           }
}
```

- Au terme de cette phase d'activité, La zone mémoire allouée pour le message est libérée.
- La zone d'émission est parcourue. Pour chaque message de cette zone d'émission, le traitement suivant est exécuté:
  - Les liens sont réellement retirés de l'environnement. Le champ *EtatEntree* est positionné à *Vide*.
  - Le message est émis vers la station de transport si le destinataire n'est pas local (fonction *EnvoiST*). Le noyau NERF conserve l'identificateur de la boîte aux lettres de la station de transport. Un message SPART est donc émis vers la station de transport.

Si le destinataire est un module local, un message SPART est émis vers la tâche TN du destinataire.

- TN détruit l'ensemble des liens de l'environnement dont la destruction avait été demandée (primitive *DestroyLink*). La fonction *DestroyNet* assure la décrémentation du compteur de duplication du module par un envoi de message.
- TN doit ensuite analyser le nouvel état du module.
  - S'il existe un quai Ouvert qui ne soit pas vide, le module (la tâche TC) peut être réactivé. La fonction *ActiverTC* réveille la tâche TC.
  - S'il n'existe plus de référence vers le module ou tous les quais du modules sont fermés. Le module de OMPHALE ne peut être réactivé. Le module devient alors **Mourant.**
  - Le passage à l'état **Mourant** déclenche l'exécution de la fonction *DevientMourant*

```
void DetruireMessagesQuai (P)
 MessageSurQuai *P ;
{
 MessageSurQuai *pc,*pp ;

 for (pc = P ; (pc - NIL) ;)
   {
    DetruireMessageRecu (AddAbsolTN (pc->PtMUtil)) ;

    pp = pc ;
    pc = pc->Suivant ;
    DisposeLocalTN (pc,sizeof(MessageSurQuai)) ;
   }
}

void DetruireZR (PCO)
 ContexteOmphale PCO ;
{
 int i = 0 ;

 for (i = 0 ; i < MAXSERVICES; i++)
   DetruireMessagesQuai (PCO->Quai [i]) ;
}

void DetruireTC (PCO)
 ContexteOmphale PCO ;
{
 status cr ;
 char MessFin [MESS_LENGTH] ;

 MessFin [0] = CodeDetruire ;
 send (MessFin,PCO->MbxIdTC,0) ;
}


void DevientMourant (PCO)
ContexteOmphale PCO ;
```

```
{
 int i ;

 PCO->Etat = Mourant ;
 DetruireTC (PCO) ;

 for (i=0;i< SizeOfEnvironnement;i++)
   if ((*(PCO->Environnement)) [i].Occupe)
     DestroyNet ((*(PCO->Environnement)) [i].LeLien.SiteName,
                 (*(PCO->Environnement)) [i].LeLien.Destinataire
                ) ;
 DisposeGlobalTN (PCO->Environnement,sizeof (Env)) ;
 DisposeGlobalTN (PCO->MessAtt,sizeof(MessageAttendre)) ;
 DetruireZR (PCO) ;
}
```

- *DevientMourant* débute par la destruction de la tâche TC. La procédure *DetruireTC* effectue cette destruction en envoyant un message stop à la tâche TC.
- Les liens se trouvant encore dans l'environnement sont détruits.
- L'espace mémoire utilisé pour l'environnement et la zone de communication TC-TN est restitué.
- La fonction *DetruireZR* détruit les messages restant dans la zone de réception. (Ainsi que les liens accompagnant ces messages).

– Lorsque le compteur de duplication est nul et que le module est mourant, le module disparait. La procédure *DetruireTN* est alors appelée.

– Lorsque le champ *Normal* est positionné à FALSE, les messages de la zone d'emission sont détruits par la fonction *DetruireMessageZE*. Le champ *QuaisOuverts* est positionné à 0. Le module va donc ainsi passer à l'état **Mourant**.

```
void DetruireMessageZE (PCO)
ContexteOmphale PCO ;
{
 MessBuffEmission *pc,*pp ;
 PtMessUtilisateur Ptu ;

 for (pc = (MessBuffEmission *) AddAbsolTN (PCO->MessAtt->Debut) ;
      (pc - NIL) ;)
   {
    Ptu = (PtMessUtilisateur) AddAbsolTN (pc->PtMUtil) ;
    DetruireMessageRecu (Ptu) ;

    pp = pc ;
    pc = (MessBuffEmission *) AddAbsolTN (pc->Suivant) ;
    DisposeGlobalTN (pp,sizeof (MessBuffEmission)) ;
   } ;
}
```

– Le traitement est est analogue à celui de la primitive *DeleteAllMessages* de l'agence Messages.

G.5 InitTN

Cette primitive de l'agence routage assure l'initialisation du contexte de la tâche TN. Elle est appelée au début de l'exécution de la tâche TN.

```
status InitTN (MessSpart)
char MessSpart [] ;
{
 unsigned int i,P ;
 ContexteTN *Cont ;
 ContexteOmphale PCO ;
 char MessSpartInit [MESS_LENGTH] ;

 Cont = (ContexteTN *) tache_courante () ;
 Cont->CtxSysOmphale = (ContexteOmphale)
                       NewLocalTN (sizeof(StructContexteOmphale)) ;
 PCO = Cont->CtxSysOmphale ;
 memcpy (&PCO->MbxIdTN,MessSpart,sizeof(xx_id)) ;
 memcpy (&PCO->MbxIdTC,MessSpart+sizeof(xx_id),sizeof(xx_id)) ;
 PCO->Environnement = NIL ;
 PCO->MessAtt = NIL ;
 PCO->QuaisOuverts = PCO->QuaisOuIlyaDesMessages = 0 ;
 PCO->Etat = EnCours ;
 PCO->DuplicateCount = 2 ;

 find (GLOBAL,"000001",&PCO->SegIdSyst) ;
 access_segment (PCO->SegIdSyst,READ_WRITE,&PCO->BaseSyst) ;

 for (i=0;i<MAXSERVICES;i++)
    {
     PCO->Quai[i].Tete = NIL ;
     PCO->Quai[i].Queue = NIL ;
    } ;

 PCO->Environnement = (Env *) NewGlobalTN (sizeof (Env)) ;

 {
  int i ;
  Lien *Ptl ;
  unsigned int Penv ;

  for (i=0 ; i < SizeOfEnvironnement ; i++)
     {
      (*(PCO->Environnement)) [i].EtatEntree = Vide ;
      (*(PCO->Environnement)) [i].Version = 0 ;
     }

  (*(PCO->Environnement)) [0].EtatEntree = LienPermanent ;

  Ptl = &((*(PCO->Environnement)) [0].LeLien) ;
  Ptl->SiteName = MyHouse ;
  Ptl->Destinataire = PCO->MbxIdTN ;
```

```
  Ptl->UseOrNo = -1 ;
  Ptl->OneOrNUse = -1 ;
  Ptl->MayDuplicate = TRUE ;

  (*(PCO->Environnement)) [1].EtatEntree = LienPermanent ;

  Ptl = &((*(PCO->Environnement)) [1].LeLien) ;
  find (GLOBAL,"MBGN",&Ptl->Destinataire) ;
  Ptl->SiteName = MyHouse ;

  MessSpart [0] = CodeDuplication ;
  send (MessSpart,Ptl->Destinataire,0) ;
  Ptl->UseOrNo = -1 ;
  Ptl->OneOrNUse = -1 ;
  Ptl->MayDuplicate = TRUE ;

  (*(PCO->Environnement)) [2].EtatEntree = LienPermanent ;

  Ptl = &((*(PCO->Environnement)) [2].LeLien) ;
  Ptl->SiteName = MyHouse ;
  find (GLOBAL,"MBGM",&Ptl->Destinataire) ;

  MessSpart [0] = CodeDuplication ;
  send (MessSpart,Ptl->Destinataire,0) ;

  Ptl->UseOrNo = -1 ;
  Ptl->OneOrNUse = -1 ;
  Ptl->MayDuplicate = TRUE ;
 }

 PCO->MessAtt = (PtMessageAttendre) NewGlobalTN (sizeof(MessageAttendre)) ;
 PCO->MessAtt->NombreLiensRecus = 0 ;
 PCO->MessAtt->Debut = PCO->MessAtt->Fin = NIL ;
 PCO->MessAtt->QuaisAOuvrir =  0 ;
 PCO->MessAtt->Longueur = 0 ;

 P = AddRelatTN (PCO->Environnement) ;
 memcpy (MessSpartInit,&P,sizeof(unsigned int)) ;

 P = AddRelatTN (PCO->MessAtt) ;
 memcpy (MessSpartInit+sizeof(unsigned int),&P,sizeof (unsigned int)) ;

 send (MessSpartInit,PCO->MbxIdTC,0) ;
 return PCO->MbxIdTN ;
}
```

- Récupération des numéros des boîtes aux lettres *MbxIdTN* et *MbxIdTC*.
- Accès au segment système. *SegIdSyst* et *BaseSyst*.
- La zone de réception du module est initialisée. Les pointeurs *Tete* et *Queue* sont initialisés à *NIL*.

- Création et allocation mémoire pour l'environnement. L'environnement est ensuite initialisé. Les trois premières entrées de l'environnement sont initialisées:
  - Le lien 0 : un lien vers le module lui-même.
  - Le lien 1 : un lien vers le gestionnaire de noms du site.
  - Le lien 2 : un lien vers le gestionnaire de modules du site.
- La valeur de duplication est initialisée à 2. En effet le module posséde une référence vers lui-même ainsi que le module qui l'a créé.
- Création de la zone de communication TN-TC (*MessAtt*). L'adresse de cette zone et de l'environnement sont communiquées à TC.

Annexe H
Les tâches systèmes

H.1 La tâche GT

La tâche GT assure la création de module. Elle effectue en particulier la création de la tâche TN du module. Il existe une tâche GT par module de traitement noyau (MTN). Cette création est effectuée à la demande de modules du CORTEX qu'on appelle le gestionnaire de module.

```
xx_id MbIdRelai, MbIdGT, SegIdGT, MbReply ;
xx_id ProgIdTN, TaskIdTN ;
int Base ;
char MessTN [MESS_LENGTH] ;

void InitGT ()
{
 find (LOCAL,"CODTN",&ProgIdTN) ;

 find (GLOBAL,"BLREL",&MbIdRelai) ;

 find (GLOBAL,"BLGT",&MbIdGT) ;

 find (GLOBAL,"SEGT",&SegIdGT) ;

 access_segment (SegIdGT,READ_WRITE,&Base) ;
}

void CreerModule (MessSpart)
 char MessSpart [] ;
{
 status cr ;
 general_address ga ;
 MessGT *PtMessGT ;

 memcpy (&ga,MessSpart,sizeof(general_address)) ;
 loc_addr (&ga,&PtMessGT) ;
 create_mbx (GLOBAL,128,&PtMessGT->mbn) ;
 create_mbx (GLOBAL,2,&PtMessGT->mbc) ;
 PtMessGT->GTDemandeur = MbReply ;
 send (MessSpart,MbIdRelai,0) ;
 receive (MessSpart,MbReply,INFINITE) ;

 if (PtMessGT->cr == OK)
   {
    create_task (ProgIdTN,20,SYS,1,&TaskIdTN) ;
    memcpy (MessTN,&PtMessGT->mbn,sizeof(xx_id)) ;
    memcpy (MessTN+sizeof(xx_id),&PtMessGT->mbc,sizeof(xx_id)) ;
    start_task (TaskIdTN,MessTN) ;
    MessSpart [0] = OK ;
```

```
    memcpy (MessSpart+1,&PtMessGT->mbn,sizeof(xx_id)) ;
   }
  else {
        MessSpart [0] = PtMessGT->cr ;
        delete_mbx (PtMessGT->mbc) ;
        delete_mbx (PtMessGT->mbn) ;
       }
 send (MessSpart,PtMessGT->ModuleDemandeur,0) ;
 free_memory (ga.seg_id,((unsigned int)PtMessGT-Base),sizeof(MessGT)) ;
}

main ()
 {
  status cr ;
  char MessSpart [MESS_LENGTH] ;

  InitGT () ;

  while (TRUE)
      {
       receive (MessSpart,MbIdGT,INFINITE) ;
       CreerModule (MessSpart) ;
      }
}
```

- La procédure *InitGT* initialise la tâche GT:
  - Le code de la tâche TN est chargé par la tâche LOTN. L'identificateur du code est récupéré via le catalogue SPART.
  - Il faut également récupérer les identificateurs de la boîte aux lettres des tâches *Relai* et celle des Tâches *GT*.
- Une **création** de module consiste à:
  - créer les boîtes aux lettres MBN et MBC.
  - créer et lancer la tâche TC du module. Cette fonction est assurée par une des tâches *Relai* du site.
  - Si tout s'est bien passé du côte de la tâche *Relai*, la tâche TN peut être créée et initialisée. Cette création de tâche se fait à partir du code de la tâche qui a été préalablement chargé par la procédure *InitGT*.
  - La tâche TN est lancée (*start_task*) en lui envoyant comme paramètre initial les identificateurs de MBN et MBC.

H.2 La tâche MODPER

La fonction de la tâche MODPER est de créer les modules permanents du système. Ces modules sont des modules de la couche CORTEX tels que GN et GM. Si cette couche CORTEX devait être étendue, certains modules seraient lancés par cette tâche.

```
#include "GT.h"
xx_id MbIdRelai, SegIdGT, MbReply ;
xx_id ProgIdTN, TaskIdTN ;
int Base ;
char MessTN [MESS_LENGTH] ;


void InitMODPER ()
{
 create_mbx (GLOBAL,4,&MbReply) ;

 find (GLOBAL,"BLREL",&MbIdRelai) ;

 find (GLOBAL,"SEGT",&SegIdGT) ;

 access_segment (SegIdGT,READ_WRITE,&Base) ;

 find (LOCAL,"CODTN",&ProgIdTN) ;
}

void LancerGN ()
{
 xx_id MbnGestNom,MbcGestNom ;
 status cr ;
 general_address ga ;
 MessGT *PtName ;
 char MessSpart [MESS_LENGTH] ;

 find (GLOBAL,"MBGN",&MbnGestNom) ;
 create_mbx (GLOBAL,2,&MbcGestNom) ;
 alloc_memory (SegIdGT,sizeof (MessGT),&PtName) ;
 PtName = (MessGT *) (Base + (unsigned int) PtName) ;
 PtName->Type = Create ;
 PtName->GTDemandeur = MbReply ;
 strcpy (PtName->proto,"GN") ;
 PtName->mbn = MbnGestNom ;
 PtName->mbc = MbcGestNom ;
 gen_addr (PtName,&ga) ;
 memcpy (MessSpart,&ga,sizeof(general_address)) ;
 send (MessSpart,MbIdRelai,0) ;

 create_task (ProgIdTN,20,SYS,1,&TaskIdTN) ;
 memcpy (MessTN,&MbnGestNom,sizeof(xx_id)) ;
 memcpy (MessTN+sizeof(xx_id),&MbcGestNom,sizeof(xx_id)) ;
 start_task (TaskIdTN,MessTN) ;
```

```
}

void LancerGM ()
{
 xx_id MbnGestMod, MbcGestMod ;
 general_address ga ;
 MessGT *PtName ;
 char MessSpart [MESS_LENGTH] ;

 find (GLOBAL,"MBGM",&MbnGestMod) ;
 create_mbx (GLOBAL,2,&MbcGestMod) ;

 alloc_memory (SegIdGT,sizeof (MessGT),&PtName) ;
 PtName = (MessGT *) (Base + (unsigned int) PtName) ;
 PtName->Type = Create ;
 PtName->GTDemandeur = MbReply ;
 strcpy (PtName->proto,"GM") ;
 PtName->mbn = MbnGestMod ;
 PtName->mbc = MbcGestMod ;
 gen_addr (PtName,&ga) ;
 memcpy (MessSpart,&ga,sizeof(general_address)) ;
 send (MessSpart,MbIdRelai,0) ;

 receive (MessSpart,MbReply,INFINITE) ;

 create_task (ProgIdTN,20,SYS,1,&TaskIdTN) ;
 memcpy (MessTN,&MbnGestMod,sizeof(xx_id)) ;
 memcpy (MessTN+sizeof(xx_id),&MbcGestMod,sizeof(xx_id)) ;
 start_task (TaskIdTN,MessTN) ;
}

main ()
 {

  InitMODPER () ;
  LancerGN () ;
  LancerGM () ;

  receive (MessSpart,MbReply,INFINITE) ;
  receive (MessSpart,MbReply,INFINITE) ;
  delete_mbx (MbReply) ;
  exit_task () ;
}
```

H.3 La tâche Relai

Cette tâche assure la création de la tâche TC d'un module OMPHALE. Cette fonction est réalisée à partir des primitives *load* et *unload* de l'agence programme ainsi que les primitives *create_task* et *start_task* de l'agence tâche. Cette tâche assure que seules les images binaires des modules nécessaires sont chargées en mémoire.

```
typedef struct NOEUD
                    {
                     char Proto [79] ;
                     xx_id MbxId,ProgId ;
                     int Nombre ;
                     struct NOEUD *Suivant ;
                    } Noeud,*PtNoeud ;

static PtNoeud Tete = NIL ;
xx_id MbxId, SegIdGT ;
int Base ;

void InitRelai ()
{
 find (GLOBAL,"BLREL",&MbxId) ;

 find (GLOBAL,"SEGT",&SegIdGT) ;

 access_segment (SegIdGT,READ_WRITE,&Base) ;
}

void Creation (PtMessGT)
MessGT *PtMessGT ;
{
 char MessSpart [MESS_LENGTH] ;
 char MessTC [MESS_LENGTH] ;
 char UnixName [32] ;
 PtNoeud P ;
 status cr ;
 xx_id ProgIdTC,TaskIdTC ;
 general_address ga ;

 for (P = Tete;
      P - NIL && strcmp (P->Proto,PtMessGT->proto) ;
      P = P->Suivant
     ) ;
 if (P==NIL)
   {
    strcpy (UnixName,"/omphale/") ;
    strcat (UnixName,PtMessGT->proto) ;
    cr = load (SYST_UNIT,UnixName,REENTRANT,&ProgIdTC) ;
    if (!cr)
       {
        P = (PtNoeud) malloc (sizeof(Noeud)) ;
```

```
        strcpy (P->Proto,PtMessGT->proto) ;
        P->ProgId = ProgIdTC ;
        P->MbxId = PtMessGT->mbc ;
        P->Nombre = 1 ;
        P->Suivant = Tete ;
        Tete = P ;
       }
     else {
           PtMessGT->cr = cr ;
           PtMessGT->Type = Reply ;
           send (MessSpart,PtMessGT->GTDemandeur,0) ;
           return ;
          }
   }
    else P->Nombre++ ;

 memcpy (MessTC,&PtMessGT->mbn,sizeof(xx_id)) ;
 memcpy (MessTC+sizeof(xx_id),&PtMessGT->mbc,sizeof(xx_id)) ;
 create_task (P->ProgId,20,0,1,&TaskIdTC) ;
 start_task (TaskIdTC,MessTC) ;
 PtMessGT->cr = OK ;
 PtMessGT->Type = Reply ;
 send (MessSpart,PtMessGT->GTDemandeur,0) ;
 return ;
}

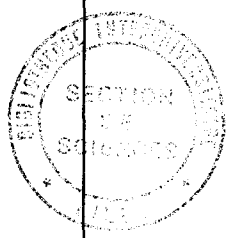
void Destruction (PtMessGT)
 MessGT *PtMessGT ;
{
  xx_id Prog ;
  PtNoeud P, Pred ;
  status cr ;

  Prog = PtMessGT->mbn ;
  for (Pred = NIL, P = Tete;
       P - NIL && (P->ProgId - Prog );
       Pred = P,P = P->Suivant
      ) ;
  P->Nombre -- ;
  if (P->Nombre == 0)
    {
     unload (Prog) ;
     if (Pred == NIL)
       Tete = P->Suivant ;
       else Pred->Suivant = P->Suivant ;
     free (P) ;
    }
}

main ()
{
  char MessSpart [MESS_LENGTH] ;
  status cr ;
  MessGT *PtMessGT ;
  general_address ga ;
```

```
  InitRelai () ;

  while (1)
      {
       receive (MessSpart,MbxId,INFINITE) ;
       memcpy (&ga,MessSpart,sizeof(general_address)) ;
       loc_addr (&ga,&PtMessGT) ;
       switch (PtMessGT->Type)
            {
             case Create : Creation (PtMessGT) ;
                           break ;
             case Delete : Destruction (PtMessGT) ;
                           free_memory (ga.seg_id,
                                        ((unsigned int)PtMessGT-Base),
                                        sizeof(MessGT)
                                       ) ;
                           break ;
             default     : free_memory (ga.seg_id,
                                        ((unsigned int)PtMessGT-Base),
                                        sizeof(MessGT)
                                       ) ;
                           break ;
            }
      }
}
```



- La tâche TC gère la réentrance des codes de modules en mémoire. Pour ce faire, elle mémorise ces codes dans une liste chaînée.
- La *Creation* de la tâche TC. La tâche *Relai* s'assure que le code de TC ne soit pas déjà chargé. Il faut pour cela parcourir la liste des codes et rechercher l'image mémoire:
  - Si la recherche est positive , une nouvelle tâche TC est crée à partir du code préalablement chargé. Le compteur d'utilisation du code est incrémenté de 1.
  - Si cette recherche n'aboutit pas, il faut donc charger le code de TC et créer une nouvelle entrée. Le compteur d'utilisation du code est initialisé à 1.

  La tâche peut ensuite être créée.
- La fonction *Destruction* est reçue lorsque l'exécution de TC est terminée (normalement ou pas). Le code de cette tâche TC est déchargé si la tâche détruite était la dernière à exécuter ce code.